
HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

# MASTERARBEIT

Herr
**Stefan Schildbach**

## Konzeption, prototypische Implementierung und Evaluierung eines Domain-Frameworks zur Anwendung computer-linguistischer Technologien auf forensische Texte

2012

Fakultät **Mathematik** / **Naturwissenschaften** / **Informatik**

# MASTERARBEIT

# Konzeption, prototypische Implementierung und Evaluierung eines Domain-Frameworks zur Anwendung computer-linguistischer Technologien auf forensische Texte

Autor:
**Stefan Schildbach**

Studiengang:
Informatik

Seminargruppe:
IF10w1-M

Erstprüfer:
Prof. Dr. rer. nat. Dirk Labudde

Zweitprüfer:
M. Sc. Michael Spranger

Mittweida, 2012

**Bibliografische Angaben**

Schildbach, Stefan: Konzeption, prototypische Implementierung und Evaluierung eines Domain-Frameworks zur Anwendung computerlinguistischer Technologien auf forensische Texte, 73 Seiten, 24 Abbildungen, Hochschule Mittweida (FH), Fakultät Mathematik / Naturwissenschaften / Informatik

Masterarbeit, 2012

Satz: LaTeX

**Referat**

*Conception, prototypical implementation and evaluation of a domain framework for the application of computational linguistic technologies on forensic texts* – Die vorliegende Arbeit beschäftigt sich mit der Konzeption eines Domain-Frameworks für die semantische Analyse von forensischen Textdaten. Die Modellierung einer Taxonomie, sowie einer Ontologie, mit Hilfe von Metadaten soll eine Recherche über einen unbekannten Datenbestand ermöglichen. Als Anwendungsdomäne werden forensische Texte betrachtet.

# I. Inhaltsverzeichnis

# II. Abbildungsverzeichnis

# III. Abkürzungsverzeichnis

| | |
|---|---|
| API | Application Programming Interface |
| CCF | Common Command Framework |
| CNF | Common Navigator Framework |
| DND | Drag & Drop |
| DSL | Domain-Specific Language |
| EMF | Eclipse Modelling Framework |
| ID | Identification (*ursprünglich: Identification Document*) |
| IDE | Integrated Development Environment |
| JDT | Java Development Tools |
| JNI | Java Native Interface |
| MDB | Master Details Block |
| MDSD | Model-Driven Software Development |
| OO | Objektorientiert |
| OSGi | Open Services Gateway initiative |
| RCP | Rich Client Platform |
| SGML | Standard Generalized Markup Language |
| SOA | Service-Oriented Architecture |
| SWT | Standard Widget Toolkit |
| UI | User Interface |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier |
| XMI | XML Metadata Interchange |
| XML | eXtensible Markup Language |

# IV. Danksagung

> „Es ist nicht gut, daß der Mensch alleine sei, und besonders nicht, daß er alleine arbeite; vielmehr bedarf er der Teilnahme und Anregung, wenn etwas gelingen soll.“
>
> *Johann Wolfgang von Goethe*

Ich möchte mich bei allen bedanken, die mich im Laufe meines Studiums, egal auf welche Art und Weise, bewusst oder auch unbewusst unterstützt haben. Ohne diese Teilnahme und die daraus resultierenden Anregungen wäre es sicherlich nicht gelungen.

Für die Zeit, in der diese Masterarbeit entstand, gilt mein besonderer Dank Herrn Professor Dirk Labudde und Herrn Michael Spranger, die mir als Betreuer stets mit offenem Ohr und kritischem Blick zur Seite standen und diese Arbeit überhaupt erst ermöglicht haben.

# 1 Einleitung

## 1.1 Motivation

Die immer größer und komplexer werdende Medienlandschaft, vor allem im Bereich des Internets und der digitalen Technik, eröffnet völlig neue Möglichkeiten der Kommunikation. War es vor einigen Jahren nur möglich sich im persönlichen Gespräch, per Telefon oder über Briefe auszutauschen, so bietet sich heute ein immer größer werdendes Feld von Kommunikationsplattformen an. Die Kommunikation erfolgt dabei vorwiegend computergestützt und auf elektronischem Wege. Das digitale Zeitalter ermöglicht es, ein riesiges Spektrum an Daten in nur kurzer Zeit mit Personen an anderen Orten zu teilen und kompakt abzulegen.

Den Überblick über die Vielzahl von Daten und die verschiedenen Ausprägungen zu behalten ist dabei nicht immer einfach. Noch schwieriger ist es, in einem großen heterogenen Datenbestand eine bestimmte Information aufzufinden bzw. wiederzufinden. Handelt es sich bei den Daten zusätzlich um fremde, nicht selbst gepflegte Daten, dann ist es doch sehr umständlich diese zu durchsuchen und zu analysieren.

## 1.2 Ziel

Computerlinguistische Technologien können derartige Aufgaben unterstützen. Diese Arbeit beschäftigt sich mit der Konzeption eines Frameworks für die semantische Analyse von Textdaten. Die Modellierung einer Taxonomie, sowie einer Ontologie, mit Hilfe von Metadaten soll eine Recherche über einen heterogenen Datenbestand ermöglichen.

Durch eine modularisierte Architektur und den Einsatz von Design-Pattern soll ein hoher Grad an Wiederverwendbarkeit und Änderbarkeit gewährleistet werden. Das Framework setzt auf der *Eclipse Rich Client Platform* (RCP) auf und kann durch verschiedene Module (Plugins) erweitert werden. Insbesondere können dadurch nachträglich neue Funktionen zur Analyse und Verarbeitung von Texten und anderen Medien hinzugefügt werden.

Als Anwendungsdomäne werden forensische Texte betrachtet. Durch die Nutzung von computerlinguistischen Verfahren können Zusammenhänge zwischen Personen, Orten und Handlungen erkannt und Hintergründe erschlossen werden. Die Generierung von neuem Wissen durch die algorithmische Verarbeitung der Texte steht dabei im Mittelpunkt. Mit Hilfe eines generierten Sprachkorpus soll die Tragfähigkeit des erarbeiteten Konzeptes evaluiert werden. Der im Rahmen der Masterarbeit entstehende Prototyp soll den Analyseprozess forensischer Texte unterstützen und beschleunigen.

## 1.3 Abgrenzung

Diese Arbeit beinhaltet die Vorstellung einer Konzeption eines Prototypen dieses Frameworks. Es werden grundlegende Strukturen und Vorgehensweisen, sowie der Ansatz einer Implementierung mit den Basisfunktionalitäten vorgestellt.

Der Prototyp ist nicht in der Lage automatische Datenanalysen vorzunehmen, diese können im späteren Verlauf der Entwicklung hinzugefügt werden. Er beinhaltet Funktionalitäten Taxonomien, Ontologien und Annotationen als Basiskonzepte, für eine weitere Verarbeitung, zu verwalten und dadurch Dokumente manuell zu klassifizieren und zu annotieren.

Durch die Entwicklung auf der Basis von Java mit der Eclipse RCP wird der Blickpunkt ausschließlich auf die Softwareentwicklung gelenkt, eine Betrachtung von Hardware und unterliegendem Betriebssystem kann vernachlässigt werden. Grundlegende Paradigmen und Kenntnisse in der Programmiersprache Java und der Objektorientierung werden vorausgesetzt und nicht weiter erläutert.

## 1.4 Aufbau

In den folgenden Kapiteln wird das im Rahmen dieser Arbeit entwickelte System vorgestellt.

Für ein besseres Verständnis der in der Arbeit verwendeten Begriffe und Methoden werden diese im nachfolgenden, zweiten Kapitel „Grundlagen“ erläutert. Darunter fallen computerlinguistische Grundlagen, welche im Projekt verwendet werden, sowie ein Einblick in die Anwendungsdomäne der forensischen Texte.

Anschließend werden vorbereitende konzeptionelle Gedanken im Kapitel „Vorbetrachtungen und Konzeption“ vorgestellt und erläutert. Es wird ein kurzer Abriss über das Zielsystem gegeben, Modellierungsgedanken für die Datenbasis vorgestellt und verwendete Kernkomponenten der Systemstruktur beschrieben.

Im Kapitel „Implementierungsdetails“ werden programmtechnische Umsetzungen näher erläutert und es wird auf Besonderheiten bei der Implementierung eingegangen. Es werden Übersichten der erstellten Systemkomponenten und Erläuterungen zu diesen vorgestellt.

Das abschließende Kapitel „Zusammenfassung und Ausblick“ liefert eine Zusammenfassung der erreichten Ziele, sowie einen Ausblick auf das mögliche weitere Vorgehen.

# 2 Grundlagen

## 2.1 Computerlinguistik

Das Fachgebiet der Computerlinguistik liegt im Überschneidungsbereich von Linguistik und Informatik und existiert seit den fünfziger Jahren. Die algorithmische Verarbeitung von natürlicher Sprache, in textueller Form oder in Form von Sprachdaten, durch den Computer bildet den Kernanwendungsbereich der Computerlinguistik. Seit ihrem Entstehen hat sie sich national und international etabliert und weiterentwickelt. Aufbauend auf dem Wissen aus der Informatik und der Linguistik wurden neue und eigenständige Methoden für die Verarbeitung, von gesprochender und geschriebener Sprache, auf maschinellem Wege entwickelt. [2, Seite 1f.]

Die Repräsentation von Sprache mittels Audioinformationen oder in der Form von Buchstabenketten ist die Ausgangslage für eine Analyse. Weitere Quellen für Sprachdaten (z.B.: Bildmaterial) können über vorgelagerte Prozesse in Textform überführt werden und stehen damit ebenfalls zur Verarbeitung bereit. Der Prozess der Sprachanalyse erfolgt in mehreren Einzelschritten, in denen man sich von der Eingangsrepräsentation schrittweise in Richtung Bedeutung vorarbeitet und währenddessen verschiedene sprachliche Repräsentationsebenen durchläuft. Dazu gehören Tokenisierung (Segmentierung des Textes in Abschnitte, Sätze, Wörter), morphologische (Rückführung der Worte auf Grundformen), syntaktische (Strukturelle Funktion des Wortes im Satz) und semantische (Zuordnung von Bedeutungen) Analyse. [28]

Text Mining ist eine computerlinguistische Verfahrensweise, in der diese Schritte umgesetzt werden. Es werden als Ausgangspunkt schwach- oder unstrukturierte Textdaten angenommen, in denen Bedeutungsstrukturen herausgearbeitet werden sollen. Der Anwender soll in die Lage versetzt werden Kerninformationen schnell aus verarbeiteten Texten zu erschließen. Im günstigsten Fall werden Informationen aufgedeckt, die dem Anwender in dieser Form nicht bekannt waren. [18]

### 2.1.1 Text Mining

Text Mining im eigentlichen Sinne ist die Anwendung des Data Mining (Aufdecken unbekannter Zusammenhänge in Daten) auf Texte. Dabei wird beim Text Mining nicht auf das Suchen, sondern auf das Finden von Informationen besonderen Wert gelegt. Beim Vorgang des Suchens wird normalerweise versucht, Wissen auf der Basis von bereits bestehenden Informationen zu beziehen. Das Problem besteht darin, alle irrelevanten Informationen bei Seite zu schieben, um die wirklich relevanten Wissensbausteine zu finden. Text Mining beschäftigt sich im Gegensatz damit, Informationen aufzudecken,

die bisher noch nicht gefunden wurden. [16]

Der Unterschied zwischen normalem Data Mining und Text Mining besteht darin, dass beim Text Mining Muster aus Text in natürlicher Sprache extrahiert werden, anstatt von strukturierten Datenbasen. Strukturierte Daten können automatisch mit Hilfe von Programmen durchsucht werden, Text hingegen ist für Menschen zum lesen gedacht und kann nicht so einfach durch Programme „gelesen“ und verstanden werden. Die Analyse von Text erfordert komplexe Sprachanalysen und statistische Methoden um Textstellen Bedeutungen zuzuweisen. [16] Der Versuch der Strukturierung oder Teilstrukturierung von Texten bzw. das Verständnis von Textkomponenten ist eine notwendige Voraussetzung für die Sprachanalyse. Mit Hilfe von semantischen Modellen können Bedeutungsträger innerhalb von Texten herausgearbeitet werden.

### 2.1.2 Semantische Modelle

Für jede erfolgreiche Kommunikation zwischen zwei oder mehr Teilnehmern ist es erforderlich, dass ein einheitliches Verständnis von Begriffen vorherrscht. Die Aufgabe, die dabei zu bewältigen ist, um einen Informationsaustausch auf diese Art und Weise und speziell mit Computern realisieren zu können, kann der täglichen Kommunikation zwischen Menschen nachvollzogen werden. Ein gegenseitiges Verständnis kann nur im Kontext der jeweiligen Vorgeschichte, der aktuellen Situation, usw. erreicht werden. Eine sprachliche Aussage oder ein verwendeter Begriff ist nur erfassbar, wenn ein umfangreiches Wissen über die Herkunft und die Hintergründe vorhanden sind. [5, Seite 64]

Es existieren verschiedene Ansätze semantische Informationen aufzubereiten und zu organisieren. Darunter zählen die Modelle der Taxonomie, des Thesaurus und der Ontologie, welche je nach Anwendungsziel verschiedene Stärken und Schwächen aufweisen. Die Mächtigkeit der Aussagekraft dieser Modelle nimmt in der Reihenfolge Taxonomie, Thesaurus, Ontologie zu. [27, Seite 2]

#### 2.1.2.1 Taxonomie

Eine Taxonomie wird verwendet, um ähnliche Objekte zu klassifizieren und in Kategorien einzuordnen – eine Hierarchie von Begriffen zu generieren. Die Ordnung von Flora und Fauna ist schon seit dem Altertum ein angestrebtes Ziel durch Biologen, was auf Grund verschiedener Ordnungsmerkmale (z.B.: Lebensraum) nicht immer eindeutig ist. Es erfolgt eine monohierarchische Anordnung der Objekte, so dass die Klassifikation als Ganzes eine Baumstruktur darstellt. Die Objekte in Wurzelnähe bilden eine Schicht von allgemeineren Informationen, wobei weiter entfernt liegende, weitverzweigte Objekte spezifischere Informationen repräsentieren. [27, Seite 3f.]

Das einleitende Beispiel der Biologen zeigt, dass die Verwendung von Taxonomien nicht nur auf die Informationstechnologie beschränkt ist und in verschiedenen Wissenschaftsdisziplinen Anwendung findet. Das Dateisystem und die darin eingebetteten Orderstrukturen von Windows zeigen eine Anwendung von Taxonomien im Bereich der Computertechnik. Begriffe oder Themen die nicht nur einem Oberelement zugeordnet werden können, müssen mehrfach angelegt werden, womit eine gewisse Redundanz entsteht. Die Taxonomie ist ein Strukturierungsmittel im Sinne von Metadaten, welches auf Grund der Schlichtheit bisher keiner bekannten Standardisierung unterzogen wurde. [27, Seite 3f.]

Die Modellierung einer Taxonomie zur Klassifizierung von Dokumenten, wird im Kapitel „3.2.4 Das Taxonomie Modell“ beschrieben.

#### 2.1.2.2 Thesaurus

Der Thesaurus kann als eine Art erweitertes Wörterbuch aufgefasst werden. Es erweitert die hierarchischen Beziehungen der Taxonomie um zwei weitere fest definierte Relationen, die Synonym- und Ähnlichkeits-Beziehung. Dadurch können Begriffe in verschiedenen Sprachen abgebildet werden und eine Multilingualität unterstützen. Weitere benutzerdefinierte Relationen sind nicht möglich. Der Thesaurus stammt ursprünglich aus dem Bibliothekarswesen und stellt eine Terminologie zu einer bestimmten Domäne dar. Es existieren unzählige Online-Thesauri die zu verschiedenen Domänen oder auch als allgemeine Thesauri angelegt wurden. Es kann eine grundlegende Unterscheidung zwischen zwei Arten von Thesauri getroffen werden, den linguistischen Thesauri zur Repräsentation des Wortschatzes einer Sprache oder einer Wissensdomäne und Thesauri zu Dokumentationszwecken. [5, Seite 92f.]

Ein linguistischer Thesaurus wird verwendet um ein Netzwerk zu erstellen, dass alle Wörter einer Zielsprache miteinander verbindet, die Zielsprache kann dabei auch eine spezifische Wissensdomäne sein. Sie werden beispielsweise verwendet um Synonyme und Antonyme bei Textverarbeitung nachzuschlagen oder automatische Erweiterungen für Suchanfragen von *Information Retrieval*-Systemen zu erzeugen. Ein Vertreter eines Thesaurus für die deutsche Sprache ist *GermaNet* (siehe `http://www.sfs.uni-tuebingen.de/lsd/`). [5, Seite 93f.] Ein Anwendungsbeispiel für den Einsatz eines linguistischen Thesaurus wird im Kapitel „3.2.5.7 Verwendung des Modells“ im Zusammenhang mit Scopes vorgestellt.

Die zweite Art der Thesauri, die Thesauri zur Dokumentation, werden verwendet um eine effiziente Einordnung von Dokumenten über ein fest definiertes Vokabular zu ermöglichen. Dadurch ist es möglich entsprechende Dokumente in eine oder mehrere Kategorien einzuordnen, die eine thematische Klassifizierung durchführen. Sie können als einfache Erweiterungen von Taxonomien betrachtet werden. [5, Seite 94]

#### 2.1.2.3 Ontologie

„Die Erfassung und Beschreibung der realen Welt, sowie des Wesens der uns umgebenden Dinge interessiert die Menschheit bereits seit Jahrtausenden: Die Ontologie — die Lehre vom Seienden — sucht als Disziplin der Philosophie seit jeher nach Möglichkeiten, die Grundstrukturen der Realität korrekt und allgemeingültig zu beschreiben.“ [5, Seite 64]

In der Computerlinguistik werden Ontologien in einer pragmatischeren Sichtweise verwendet. Das Ziel, ein effektive und formale Kommunikation für ein gemeinsamen Verständnis, kann mit einer gemeinsamen Konzeptualisierung, für die Kommunikationsteilnehmer, als Grundlage für die Kommunikation geschaffen werden. Dabei wird das notwendige Wissen über Begriffe und Zusammenhänge für eine erfolgreiche Kommunikation formal beschrieben. Formal heißt in diesem Zusammenhang, dass die Darstellung in maschinenverständlicher Form vorliegt. Eine Ontologie ist eine Art der Repräsentation von Wissen, die für diese Aufgabe verwendet werden kann. [5, Seite 64]

Man unterscheidet bei Ontologien zwischen zwei Graden der Allgemeingültigkeit. Eine *Domain-Ontologie (oder domänenspezifische Ontologie)* modelliert eine spezifische Domäne, welche einen Teil der Welt repräsentiert. Besondere Bedeutungen von Begriffen, die innerhalb dieser Domäne auftreten, werden durch die Domain-Ontologie beschrieben. Eine *Grundlagen-Ontologie* bildet ein Modell der gemeinsamen Objekte, die auf einem breiten Spektrum von Domain-Ontologien allgemein anwendbar sind. Sie bilden ein Glossar aus Grundbegriffen und Objektbeschreibungen, wie sie in verschiedenen relevanten Domänen verwendet werden. [5, Seite 67] [19]

Die am häufigsten gebrauchte, allgemeingültige Definitionen von Ontologie in Bezug auf die Informatik, „Eine Ontologie ist eine formale, explizite Spezifikation einer gemeinsamen Konzeptualisierung“, wurde in den 90er Jahren durch Gruber, Uschold und Gruninger geprägt. [5, Seite 64f.]

„Im Grunde genommen geht es also um: die Nutzung gemeinsamer Symbole und Begriffe im Sinne eines Syntax, das gemeinschaftliche (Ein-)Verständnis bezüglich deren Bedeutung, also der Semantik, die Klassifikation der Begriffe in Form einer Taxonomie, die Vernetzung der Begriffe mit Hilfe von assoziativen Relationen bei gleichzeitiger Festlegung von Regeln und Definitionen darüber, welche Relationen sinnvoll und erlaubt sind.“ [5, Seite 65]

Ein Anwendungsbereich für Ontologien findet sich in der Informationsextraktion, bei der Strukturen, Formalisierungen von Sprachkonzepten oder formale Beschreibungen der Anwendungsdomäne der Ontologie den Extraktionsprozess unterstützen können. Die *Extraktionsontologie* stellt eine besondere Integrationsform für Ontologien dar, die zur

Modellierung von Templates[1] (im Sinne der Informationsextraktion), durch Techniken der Wissensrepräsentation, verwendet werden kann. Als größter Nachteil von Extraktionsontologien ist der Aufwand bei der Generierung zu nennen. Eine weitere Integrationsform von Ontologien ist die formale Modellierung von Domänenwissen, bei dem formale Beschreibungen existierender Konzepte verwendet werden können, um zusätzliches Wissen bei der Extraktion von Informationen heranzuziehen. Dies kann als eine Art Brücke oder Verbindung zwischen der realen und formalen Welt verwendet werden. [5, Seite 219ff.] Im Rahmen der Arbeit werden Extraktionsontologien zur Informationsextraktion aus Texten verwendet.

Zusammenfassend ist für eine funktionierende Kommunikation auf formalem Wege notwendig, ein abstraktes Modell zu erstellen, dass relevante Begriffe und deren Beziehungen zueinander erfasst. Alle Kommunikationspartner müssen diese Begriffe akzeptieren, damit sie gemeinsam benutzt werden können und ein gemeinsamer Kommunikationskontext entstehen kann. Da Ontologien, im informationstechnischen Sinn, für abgegrenzte Interessengebiete mit gegebenenfalls ausgewählten Zielsetzungen, formuliert werden, wird im Gegensatz zum philosophischen Ursprung von mehreren unterschiedlichen Ontologien, je nach Domäne oder Zielsetzung, gesprochen. [5, Seite 65]

Es lässt sich erkennen, dass es sich bei Thesauri und Taxonomien – unter Einhaltung bestimmter Konventionen – um einfache Ontologien handelt. Die Repräsentation von Ontologien kann über verschiedene Modelle realisiert werden, die unterschiedlichen Mächtigkeiten unterliegen.

**Concept Map**

„Concept Mapping ist eine Methode zur Wissensstrukturierung und Visualisierung, die in den 60er Jahren des 20. Jahrhunderts von Josef Novak an der Cornell Universität entwickelt wurde. Eine Concept Map (Konzeptlandkarte) im Sinne von Novak ist eine abstrakte Beschreibung bestimmter Ideen oder einer Wissensdomäne [...].“ [5, Seite 80]

Concept Maps visualisieren semantische Einheiten (Präpositionen) über einer bestimmten Domäne. Semantische Einheiten bestehen aus zwei Begriffen (Konzepten), die mit einer benannten Relation verbunden sind, wobei es auch erlaubt ist Querrelationen zwischen Konzepten zu erstellen. Relationen können gelabelt werden um Ihnen eine Semantik zu zuweisen. [5, Seite 81ff.]

**Topic Map**

„Die Topic Map [(Themenlandkarte)] besteht im Wesentlichen aus Subjects (abstrakte Dinge), Topics (deren adressierbare Repräsentation), Assoziationen, Scopes (Gültigkeitsbereiche für Topics) und zugeordneten Dokumenten außerhalb der Topic Map

[1] In der Informationsextraktion beschreibt ein Template die zu extrahierende Entitäten anhand ihrer Eigenschaften.

(Occurences).“ [27]

Die Topic Map ist das in der Literatur am besten definierte Konzept unter den bisher vorgestellten Konzepten, da es durch die ISO Standardisiert wurde. Es existieren weitere offene Standards (z.B.: XML-Standard), die die Grundkonzepte der Topic Map umsetzen, aber einzelne Aspekte der ISO-Norm vernachlässigen oder modifizieren. [27]

Eine Topic Map ist ein durch die ISO/IEC 13250[2] beschriebener Standard, der den Austausch, die Organisation und die Repräsentation von Informationen und Wissen mit Hilfe von Topics (engl. Themen) ermöglicht. [17]

Im Allgemeinen haben die strukturellen Informationen, die von einer Topic Map bereit gestellt werden, die Möglichkeit Beziehungen zwischen Topics zu beschreiben und adressierbare Informationsobjekte um Topics herum anzulagern (Occurrences). Es ist möglich Topics mit speziellen Eigenschaften (Facets) zu versehen und die einzelnen Informationsbausteine in einen Kontext bzw. einen Gültigkeitsbereich einzuordnen. [17]

Mehrere Topic Maps können strukturelle Informationen zu der selben Informationsressource bereitstellen. Die Architektur ist dabei so ausgelegt, dass verschiedene Topic Maps zusammengeführt und Informationen zu verschiedenen Bereichen gekoppelt werden können. Auf Grund ihres extrinsischen Charakters können Topic Maps als Überlagerung oder Erweiterung für Informationsobjekte angesehen werden. [17]

Topic Maps ermöglichen vielseitige und gleichzeitige Betrachtungsmöglichkeiten für Informationsobjekte. Die strukturelle Natur dieser Möglichkeiten ist uneingeschränkt. Es ist möglich einen objektorientierten, relationalen, hierarchischen, sortierten oder unsortierten Ansatz, aber auch eine Kombination aus diesen zu wählen. Zusätzlich können unbegrenzt viele Topic Maps für eine gegebene Menge an Informationsressourcen übereinander gelegt werden. [17]

In Kapitel „3.2.5 Das Ontologie Modell“ wird ein konkretes Modell für die Repräsentation einer Ontologie mit Hilfe von Topic Maps vorgestellt.

### 2.1.3 Zusammenfassung

Die in diesem Kapitel erwähnten Modelle eignen sich alle zur Strukturierung und Repräsentation von Informationen und Wissen. Während Taxonomien und Thesauri eher als Hilfsmittel in Anwendungen eingesetzt werden, sind Concept Maps und Topic Maps, beziehungsweise Ontologien, für anspruchsvollere Aufgaben geeignet. Durch die Verwendung von Ontologien, als eine strukturelle und funktionelle Erweiterung gegenüber Taxonomien und Thesauri, können Wissensnetzwerke erstellt werden, die zukunftsfä-

[2] Vergleiche auch [17]

hig, flexibel und erweiterbar sind. [27]

Die Evaluierung der Meta-Konzepte Concept Map und Topic Map, zur Erzeugung einer Ontologie führten zum Schluss, dass die Verwendung von Topic Maps zwar komplizierter zu handhaben ist, aber in der Aussagekraft deutlich vor den Concept Maps liegt. Auf den ersten Blick erlauben Concept Maps eine enorme Freiheit und Aussagemächtigkeit für den Ersteller. Jedoch sind Concept Maps, zur formalen Repräsentation des Wissens bei einer engeren Betrachtung nicht geeignet, weil sie erhebliche Schwächen in der Modellierungsfähigkeit von komplexeren Strukturen aufweisen.

## 2.2 Forensik

Der Begriff Forensik umfasst wissenschaftliche Arbeitsgebiete, die sich mit der systematischen Identifizierung, Analyse und Rekonstruktion von kriminellen Handlungen beschäftigen. Ein spezielles Arbeitsgebiet, auf das in dieser Arbeit ausschließlich bei der Referenzierung des Begriffs Forensik Bezug genommen wird, ist die IT-Forensik. Die IT-Forensik (oder auch Computer Forensik, Digitale Forensik) beschäftigt sich mit der Erfassung, Aufbereitung und Analyse digitaler Spuren, welche im Zusammenhang mit Ermittlungen und Straftaten im Bereich der Computerkriminalität sichergestellt bzw. beschlagnahmt werden. [14, Seite 1f.]

**Forensische Linguistik**
Als Anwendungsgebiet für die vorliegende Arbeit werden forensische Texte betrachtet, diese liegen im Arbeitsbereich der linguistischen Forensik (oder auch forensischen Linguistik, textuellen Linguistik). Die linguistische Forensik versucht den Bereich der Sprache mit dem Bereich des Rechts zu vereinen – die Verbindung des Sprachlichen mit den entsprechenden rechtlichen Folgerungen. „Forensische Linguistik ist ein Teilgebiet der Linguistik, der die linguistische Analyse solcher sprachlicher Daten (einschließlich ihrer Präsentation vor Gericht) umfasst, die Gegenstand juristischer Betrachtungen sind.“ [9, Seite 15f.]

Die Forschungsfelder für diese spezielle Forensik sind weit gestreut und vielseitig, sie sind dabei aber nicht immer eindeutig voneinander zu trennen. Eine mögliche Untergliederung kann in die Sprache juristischer Texte, sprachliche Phänomene als Gegenstand juristischer Betrachtung und linguistische Gutachten zu Autor- und Täterschaft stattfinden. Im Speziellen fallen darunter die Analysen von Wortbedeutungen, Ähnlichkeiten, Markenrecht, Beleidigungen, Autorenschaft, etc. [9, Seite 16f.]

Ein generelles Arbeitsfeld der forensischen Linguistik ist die Analyse der Äußerungsbedeutung, die es zur Aufgabe hat, ein Verständnis von betreffenden Äußerungen, die in Form von Text oder Sprache vorliegen, zu schaffen. Dabei wird auf die Verwendung von semantischen und pragmatischen Analyseverfahren zurückgegriffen. Der Schwerpunkt

– die Rekonstruktion des Verständnisses – ist in den meisten Fällen, auf Grund von fehlenden semantischen Einheiten, ein schwieriges Unterfangen. [9, Seite 22] Die Anzahl der fehlenden semantischen Einheiten können in der Anwendungsdomäne zusätzlich durch Fachausdrücke, fehlerhafte oder verschleierte Texte oder Dialekte weiter erhöht werden. Ein erster Ansatzpunkt zur Milderung dieser Problemstellung wird im Kapitel „3.2.5.6 Verwendung von Scopes“ vorgeschlagen.

Das Hauptbetrachtungsfeld der im Rahmen dieser Arbeit entstehenden Anwendung, liegt in der Analyse der Äußerungsbedeutung und der Feststellung der Täter- und Autorschaft.

**Aktueller Stand der Technik**
Im Falle einer kriminalistischen Untersuchung werden häufig Computer und andere Medien zur Datenspeicherung sichergestellt. Es ist notwendig diese sichergestellten bzw. beschlagnahmten textuellen Daten während der Evaluierung verschiedenen Analysen zu unterziehen. Die Texte müssen fallbasiert nach kriminalistischer Relevanz gefiltert werden, es muss nach Entitäten und Beziehungen innerhalb und zwischen den Texten gesucht, Beziehungsnetzwerke und geplante Aktivitäten aufgedeckt, versteckte semantische Einheiten gefunden und verstanden werden – es müssen beweiskräftige Informationen gesucht und gesichert werden können. Derzeit verwendet man verschiedene, hochgradig spezifizierte Lösungsansätze für spezielle Problemstellungen, welche größtenteils einer manuellen Evaluierung unterliegen. Auf Grund der Aufgabenfülle und der Menge an Daten, benötigt die Evaluierung im Falle von Wirtschaftskriminalität durchschnittlich sechs Monate und mehr. [3]

Eine Verbesserung der aktuellen Situation kann auf verschiedene Art und Weisen realisiert werden. Darunter fällt die Integration bestehender Software-Lösungen, welche diese Aufgaben innehaben, eine einzelne computergestützte Lösung, welche die Handarbeit der Evaluierung ersetzt und ein Expertensystem, welches forensische Texte handhabt. [3]

Das in dieser Arbeit vorgestellte Projekt befasst sich mit der Entwicklung eines Anwendungsrahmens zur Bereitstellung einer Toolbox, welche die Aufgaben der Evaluierung von forensischen Texten im Gebiet der linguistischen Forensik aufgreift. Die Toolbox umfasst verschiedene computerlinguistische Technologien, die bei diesen Aufgaben unterstützend eingesetzt werden können und die derzeit fast ausschließlich manuell durchgeführten Analyseprozesse der forensischen Linguistik algorithmisch aufwerten.

[3] Vergleiche auch [24]

# 3 Vorbetrachtungen und Konzeption

## 3.1 Einordnung

Die zu entwickelnde Anwendung soll mittels computerlinguistischer Technologien IT-forensische Daten (z.B.: Texte, Bilder, Audio, ...) erfassen und analysieren. Eine spätere Erweiterung und Modifikation durch verschiedene Module soll gewährleistet werden.

Ein besonderes Augenmerk wird dabei auf die Annotation von Dokumenten gelegt. Im Detail sollen dabei Beziehungen zwischen Personen herausgearbeitet, Texte bestimmten Personen zugeordnet und Aktivitäten erkannt werden. Als weitere Funktionalität wird das Erkennen und Verstehen von fehlerhaften Texten, sowie gruppenspezifischen Sprachen und Dialekten erarbeitet.

Die in dieser Arbeit beschriebene Architektur bezieht sich auf den Prototypen des Systems, welches noch nicht mit vollständiger Funktionalität ausgestattet ist und sich vorerst auf die Generierung eines modularen Frameworks mit Möglichkeiten der Dokumenten-Annotation konzentriert.

Die Abbildung 3.1 stellt das Zielsystem im Kontext seiner Anwendung dar. Es sollen Datenstrukturen aus verschiedenen Quellen extrahiert und in das System eingespeist werden können, welches diese analysiert, verarbeitet und aufbereitet.

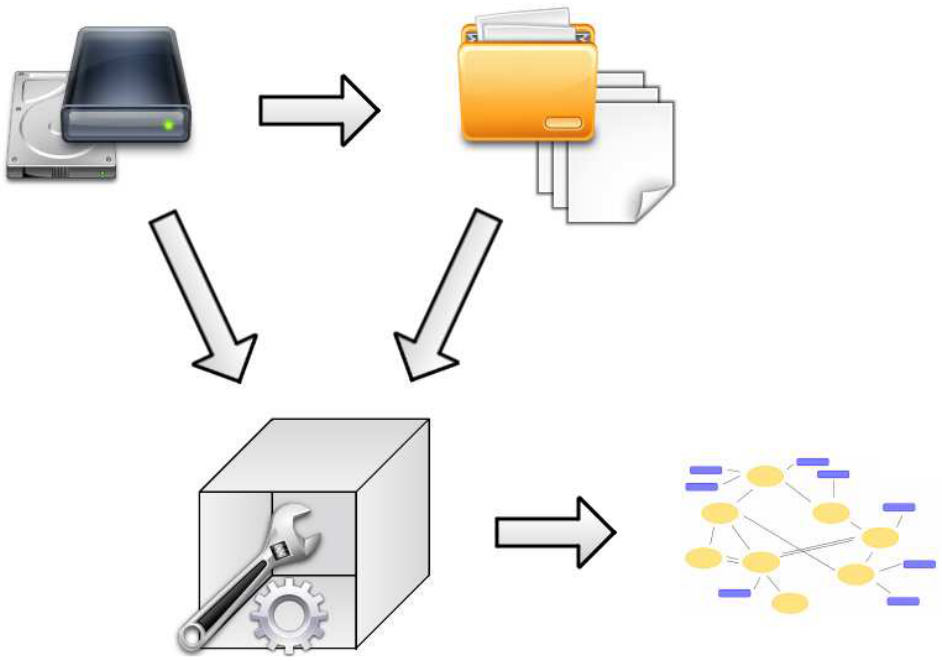

Abbildung 3.1: Das Zielsystem im Kontext seiner Anwendung

**Einsatz des Prototyp**

Das System dient zur Unterstützung des Analyse- und Auswertungsverfahrens von IT-forensischen Daten. Zielgruppe des Systems sind die Mitarbeiter mit analytischer Tätigkeit in den Kriminalpolizeiinspektionen des Landes Sachsen. [4] Die Einteilung der Mitarbeiter in Rollenschemata erfolgt dabei in drei Ausprägungen.

Ein technischer Administrator verwaltet die Nutzer des Systems und legt die Grundstrukturen fest, die zur Nutzung des Programmes benötigt werden. Eine Rolle „Erweitert“ verfeinert die Grundstrukturen, um notwendige Detaillierungsgrade für einen speziellen Anwendungsbereich zu schaffen. Die Rolle „Basis“ speist Daten in das System ein und bearbeiten diese. Es ist ihr möglich Anfragen an das System zu stellen. Der Rolle „Erweitert“ werden zusätzlich alle Funktionalitäten zur Verfügung gestellt, die die Rolle „Basis“ besitzen.

Datenschutzrechtliche Aspekte sollen vom System respektiert werden, insbesondere Löschfristen. Eine Übersicht über alle Anwendungsfälle ist im Anhang in der Abbildung A.3 hinterlegt.

**Informationsgewinnung**

Die Originaldaten werden mit Hilfe von computerlinguistischen Technologien analysiert und aufbereitet. Im Anschluss erfolgt eine Annotation der gewonnenen Daten, wobei ausschließlich auf den extrahierten Daten gearbeitet wird, die Originale bleiben unverändert. Um einen Bezug zwischen Originaldaten und extrahierten Daten herstellen zu können, wird eine Referenz (*Back Reference*) auf die Originaldaten mit abgelegt. Dem Anwender ist es möglich Anfragen an das System zu stellen. Die Abbildung 3.2 beschreibt diesen Prozess der Informationsgewinnung.

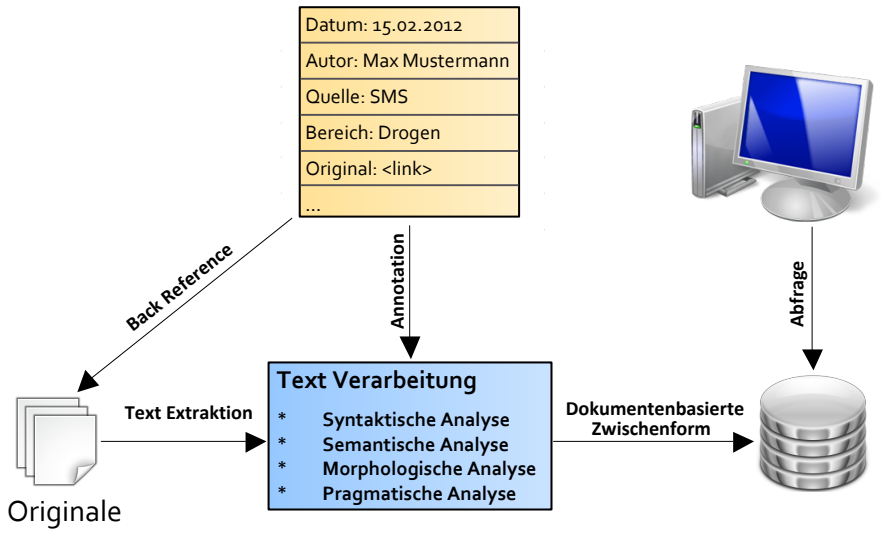


Abbildung 3.2: Prozess der Informationsgewinnung

[4] Als Ansprechpartner für fachliche Rückfragen stehen die Mitarbeiter der Kriminalinspektion Chemnitz/Erzgebirge zur Verfügung.

Bei der Annotation werden Daten mit zusätzlichen Informationen angereichert, die Hintergründe offen legen, Einordnungen erlauben oder andere Wissensbausteine liefern können. Die Anwendung soll mit Hilfe von zu definierenden Eigenschaften Daten beschreiben können und nach vorgegebenen Mustern Strukturen extrahieren. Um eine Annotation von Daten durchführen zu können, müssen verschiedene Aufgaben schrittweise bearbeitet werden: Die Erzeugung einer Taxonomie als Klassifizierungs-Template für Dokumente, die Erzeugung einer Ontologie als Template und damit verbunden einer DSL zur Repräsentation des Sprachraumes, die Einordnung von Dokumenten in ein Taxonomie-Template und die gleichzeitige Extraktion von Dokumenten-Daten mit Hilfe eines Ontologie-Templates, sowie die Generierung eines Index zur Suche über den annotierten Datenbestand.

**Einordnung in das Gesamtsystem**

In Abbildung 3.3 wird ein möglicher Aufbau des Gesamtsystems dargestellt, bei dem verschiedene Benutzerrollen (hier genannt: *agent*, *division head* und *administrator*) auf unterschiedliche Bereiche des Systems zugreifen können. Die einzelnen Systembereiche (*Querying Machine*, *Indexing Machine*, *Ontology Machine*) nutzen von einander separierte Datenquellen (*original data*, *index data*, *metadata*). Der Prototyp deckt, wie schon erwähnt, nur Grundfunktionalitäten innerhalb der zu erstellenden Anwendung ab – die in der Abbildung 3.3 ausgewiesene *Ontology Machine* ist als Arbeitspaket für den Prototyp vorgesehen.

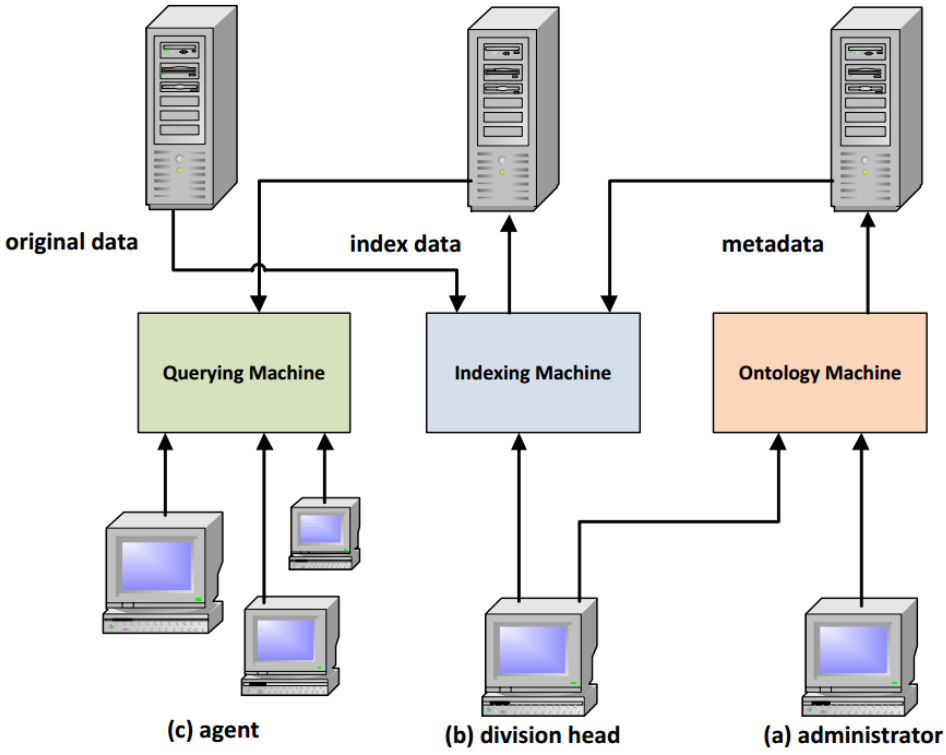


Abbildung 3.3: Einordnung in das Gesamtsystem [24]

## 3.2 Datenmodell

Für die Umsetzung der Zielstellungen des Systems, ist die Erstellung einer Ontologie mit deren Hilfe semantische Beziehungen innerhalb und zwischen einer Menge von Dokumenten erschlossen werden können notwendig. Die Annotation der Dokumente mittels der Ontologie, sowie eine Einordnung der Dokumente in eine Taxonomie stellen weitere Anforderungen in der Umsetzung.

### 3.2.1 Vorbetrachtungen

Ein Datenmodell (oder auch Domain-Modell) repräsentiert die Daten auf und mit denen man arbeiten möchte. Ein allgemein empfohlener Ansatz ist es, das Modell unabhängig zur Anwendungslogik zu entwerfen. Dies führt zu Klassen mit nahezu keiner Logik und vielen Eigenschaften. [33]

#### 3.2.1.1 Modellgetriebene Softwareentwicklung

Der Ansatz der Codegenerierung aus einem Modell heraus entspricht dem Paradigma der modellgetriebenen Softwareentwicklung (MDSD). Von modellgetriebener Softwareentwicklung spricht man, wenn Software teilweise oder vollständig aus formalen Modellen heraus generiert wird. Bei MDSD geht es darum, sich bei der Entwicklung von Softwaresystemen möglichst nicht zu wiederholen (DRY-Prinzip) und die Möglichkeit nutzen zu können, eine parallele Entwicklung und Integration von Modellen und Funktionen durchzuführen. Es werden Zielsprachen-unabhängige Abstraktionen zur Beschreibung verschiedener Sachverhalte eines Softwaresystems in Form von domänenspezifischen Sprachen (DSL) erschaffen. Diese werden dann entweder generativ oder interpretativ auf die Zielplattform abgebildet. [25, Seite 3ff., Seite 115]

Es existieren verschiedene Tools, die eine modellgetriebene Softwareentwicklung unterstützen. Ein weit verbreitetes, auf UML basierendes Tool der Firma *microTool GmbH* aus Berlin ist „objectiF“. Es bietet Möglichkeiten der Modellierung von Geschäftsprozessen und fachlichen Modellen, die über Modelltransformationen in technische Modelle umgesetzt werden können. Ein weiteres Tool bildet das quelloffene Java-Framework „Eclipse Modeling Framework (EMF)“. Beide Tools können in die Entwicklungsumgebung Eclipse, welche aus verschiedenen Gründen, auf die im Kapitel „3.3.1 Eclipse“ näher eingegangen wird, als bevorzugte Umgebung verwendet wird, integriert werden. Durch die vollständige Integration von EMF in Eclipse und die gemeinsame Community der beiden Frameworks, wird EMF gegenüber objectiF und anderen Frameworks bevorzugt, welche nur eine teilweise Integration in die Entwicklungsumgebung ermöglichen.

#### 3.2.1.2 Entwicklungswerkzeuge des Datenmodells

Das Eclipse Modeling Framework (EMF) ist eine stark Community getriebenes Projekt, welches die modellgetriebene Softwareentwicklung im Fokus hat. Dabei unterscheidet EMF zwischen einem Meta-Modell, welches die Struktur beschreibt, und dem eigentlichen Modell, welches die Instanz des Meta-Modells bildet. Es ist möglich das Meta-Modell über verschiedene Arten zu erzeugen, darunter fallen XMI, Java Annotationen, UML und XML Schema. Sobald das Meta-Modell spezifiziert wurde, können die entsprechenden Klassen als Java-Implementierung generiert werden, wobei die Möglichkeit besteht, dass der generierte Quellcode manuell erweitert werden kann. Das Eclipse Modeling Framework bietet ein austauschbares Framework an, um Modell-Informationen abzulegen. Standardmäßig wird XMI verwendet um die Modelldefinition zu persistieren, es können aber auch jegliche andere Formate zur Persistierung verwendet werden. [33]

Das Meta-Modell besteht im eigentlichen aus einem zweistufigen Modell, dem *Ecore*-Modell als Grundmodell und dem mit Informationen angereicherten *Genmodel*-Modell, welche ähnlich dem Decorator-Pattern aufgebaut sind. Das *Ecore*-Modell beinhaltet Informationen über die definierten Klassen und Eigenschaften und modelliert dabei eine domänenspezifische Sprache (DSL). Es erlaubt die Verwendung verschiedener Elemente, darunter fallen *EClass* (eine Klasse), *EAttribute* (ein Attribut mit Name und Typ), *EReference* (eine Referenz auf eine Klasse) und *EDataType* (der Datentyp eines Attributs). Die Besonderheiten dieser Objekte ist die Ableitung von dem Interface *EObject*, welches eine reflektive API zur Manipulation der Daten bereit stellt. Weiterhin ist *EObject* abgeleitet von *Notifier*, welches Modelländerungen an mögliche Interessenten über ein Observer Design Pattern weiterreicht. Das *Genmodel*-Modell verfügt über zusätzliche Informationen für die Quellcode-Generierung (z.B.: Pfade und Dateiinformationen) und Parameter zur Steuerung wie der Quellcode erzeugt werden soll. [33] [26, Seite 11ff.]

Zusätzlich zu den reinen Modell-Klassen, kann das *Genmodel* noch andere Funktionen automatisch in festgelegten Paketen erzeugen, um Standardimplementierungen zur Erzeugung und Bearbeitung der Modelle durch Wizards und Editoren, sowie Testfälle bereit zu stellen. Darunter fällt der „Edit Code“, der Basis-Adapter und -Provider (*ItemProvider*) für die Modell-Elemente erstellen kann, die durch andere Provider (z.B.: EditingDomain, ContentProvider, LabelProvider, ...) adaptiert werden um die Funktionalität, Struktur und Abhängigkeit der Elemente erfassen zu können. [26, Seite 41ff.]

Bei der Generierung des Quellcodes, der Erzeugung eines konkreten Modells – einer inneren DSL[5], werden verschiedene Klassen für ein Modell-Element erzeugt. Es wird ein Interface mit dem Element-Namen erzeugt, welches alle Signaturen der Modell-Methoden bereitstellt und das *EObject*-Interface erweitert. Die konkreten Implementierungen der Modell-Elemente findet in einer Klasse mit dem Postfix „Impl“ am Modell-

[5] Eine innere DSL beschreibt einen Ausschnitt aus dem bestehenden Vokabular der äußeren DSL.

Namen statt, die die Klasse *EObjectImpl* erweitert und das Element-Interface implementiert. Für jedes Element wird zusätzlich noch ein *ItemProvider* durch den „Edit Code“ erzeugt. Alle *ItemProvider* eines Modells können über die *ItemProviderAdapterFactory* generiert und verwendet werden. Die komplexe generierte Struktur, die von EMF vorgegeben wird, erlaubt eine hohe Flexibilität bei der Umsetzung des Modells und ist trotz ihrer Komplexität durch die automatische Erzeugung optimal aufeinander eingestellt.

Die Vorteile bei der Nutzung von EMF gegenüber anderen Modellierungen ist die Integrationsfähigkeit in die Eclipse Werkzeugplattform. Die explizite Modellierung hilft beim Verständnis des Modells. EMF erzeugt Schnittstellen und Factories zur Erzeugung von Modell-Objekten, die die Anwendung von einzelnen Implementierungsklassen sauber hält. Weiterhin ist es möglich den Quellcode jederzeit erneut vom Modell erzeugen zu lassen und Modelländerungen ohne großen Aufwand in das Modell einzubringen ohne manuelle Erweiterungen verwerfen zu müssen. [33]

### 3.2.2 Modell-Manipulation

Die Manipulation oder Veränderung von Daten innerhalb der EMF-Welt kann mit Hilfe normal generierter Getter- und Setter-Methoden erfolgen oder mit Hilfe des Command Frameworks. [6] Das Command-basierte Editieren von EMF-Objekten liefert neben einem vollständig automatisiertem Undo und Redo noch weitere Vorteile gegenüber der herkömmlichen Methodik. Dabei teilt sich das Command Framework von EMF in zwei Teile, das Common Command Framework (CCF) und die EMF.Edit Commands. [26, Seite 54ff.]

Das CCF stellt grundlegende Schnittstellen und Klassen bereit um Modelländerungen durchzuführen. Dabei ist es nicht relevant um was für ein Modell es sich handelt, EMF oder nicht. Mit Hilfe eines *CommandStacks* können Commands verwaltet und ausgeführt werden. Commands können auf diesem Stack hintereinander gestapelt und ausgeführt werden. Um nicht jede atomare Änderungen einzeln durchführen zu müssen, können mehrere einzelne Commands zu einem *CompoundCommand* zusammengefasst werden. Dieses zusammengesetzte Command delegiert etwaige Funktionen `undo()`, `redo()`, `execute()`, etc. an die einzelnen Bestandteile. Zusätzlich besitzt es weitere Methoden, wie die Funktion `appendAndExecute()`, mit der mehrere Aktionen aufgezeichnet werden und im späteren Verlauf gemeinsam wieder rückgängig gemacht werden können. [26, Seite 55ff.]

Die auf das CCF aufgesetzt und für EMF spezifizierte EMF.Edit Commands bieten zusätzlich Funktionen zur Modifikation von EMF-Objekten. Es macht sich die reflektive API der *EObject*-Klasse zu Nutze. Damit ist es möglich Attribute oder Referenzen (*Set-*

[6] Das Command Framework zu Datenmanipulation steht in keiner Beziehung zur Verwendung von Commands und Actions innherlab der UI (UI Commands). Die Verwendung des Begriffs Command in den folgenden Abschnitten bezieht sich ausschließlich auf EMF Commands.

*Command*) in EMF-Objekten zu setzen, Listenoperationen auf Eigenschaften anzuwenden (*AddCommand*, *RemoveCommand*) oder Objekte zu repositionieren (*MoveCommand*, *ReplaceCommand*, *CopyCommand*). Die elementaren Funktionen unterstützen vollständig die Umsetzung von Undo und Redo und können zur Erzeugung weiterer High-Level CompoundCommands genutzt werden. [26, Seite 59ff.]

Die Verwaltung der Modell-Ressourcen, sowie der Command-basierten Modifikation des Modells werden durch eine *EditingDomain* gehandhabt. Diese ermöglicht es zentral Commands zu erzeugen, den CommandStack zu verwalten und bietet den Zugriff auf die unterliegenden EMF Ressourcen (*RessourceSet*) an. [26, Seite 61f.]

### 3.2.3 Modell-Persistierung

EMF beinhaltet ein leistungsfähiges Framework zur Modell-Persistierung. Standardmäßig wird eine Serialisierung mittels XML und XMI über ein hochgradig anpassbare Ressourcen-Implementierung angeboten. Darüber hinaus stellt das Framework eine API bereit, die flexibel genug ist um jegliche Art der Serialisierung zu unterstützen. Es ist sogar möglich verschiedene Speicherarten zu wählen und die Objekte dennoch untereinander zu referenzieren. [26, Seite 443ff.]

Die Grundlage der Persistenz in EMF bildet eine sogenannte „*Resource*“ (Ressource), die als Container angesehen werden kann, in dem mehrere Objekte zusammen gespeichert werden können. Zur Persistierung von EMF-Objekten werden die entsprechenden Objekte zur Liste der Ressourcen-Inhalte (Contents) hinzugefügt. Beinhaltet das Objekt weitere Objekte, werden diese automatisch mit abgelegt. Die eigentliche Persistierung erfolgt final mit dem Aufruf der Schnittstellen-Mothode `save()`. Um ein aktives Objekt aus der Ressource zu laden, muss als erstes die `load()`-Methode aufgerufen und anschließend kann auf die Liste der Inhalte zugegriffen werden. [26, Seite 443ff.]

Die Handhabung von Referenzen zwischen Objekten in verschiedenen Ressourcen wird über das Interface „*ResourceSet*“, einem Container für Ressourcen, gehandhabt. Normalerweise wird eine Ressource über ein ResourceSet erzeugt und mit Hilfe einer URI identifiziert. Mit Hilfe des Schemas oder der Dateiendung der URI können unterschiedliche Fabrikmethoden angesteuert werden, die im Vorfeld registriert werden können. Das ResourceSet erzeugt die Ressource indem es die jeweiligen Fabrikmethoden ansteuert und somit kann an dieser Stelle die Art der Persistierung bestimmt werden. So kann die URI auf einen Ort im Dateisystem, im Internet oder eine Datenbank zeigen. [26, Seite 443ff.]

### 3.2.4 Das Taxonomie Modell

Eine Taxonomie[7] wird in diesem Projekt zur Klassifizierung von Dokumenten benötigt. Dieser Abschnitt erläutert das erstellte Modell näher, indem die Kernelemente *CategoryDescriptor* und *DocumentDescriptor* vorgestellt werden.

#### 3.2.4.1 Begriffsreferenzen

Eine kurze Übersicht und Beschreibung der im Folgenden auftretenden Begrifflichkeiten bezüglich des Modells.

**DomainTaxonomy** Die Klasse *DomainTaxonomy* bildet einen Container, welcher alle Dokumente beinhaltet und die Wurzel-Kategorie festlegt.

**CategoryDescriptor** Ein *CategoryDescriptor* stellt ein repräsentatives Element für eine Kategorie dar.

**DocumentDescriptor** Jedes Dokument wird innerhalb der Taxonomie auf einen *DocumentDescriptor* abgebildet.

**DocumentProperty** Einem *DocumentDescriptor* können mehrere zusätzliche Eigenschaften mit Hilfe einer *DocumentProperty* zugewiesen werden.

#### 3.2.4.2 Modellübersicht

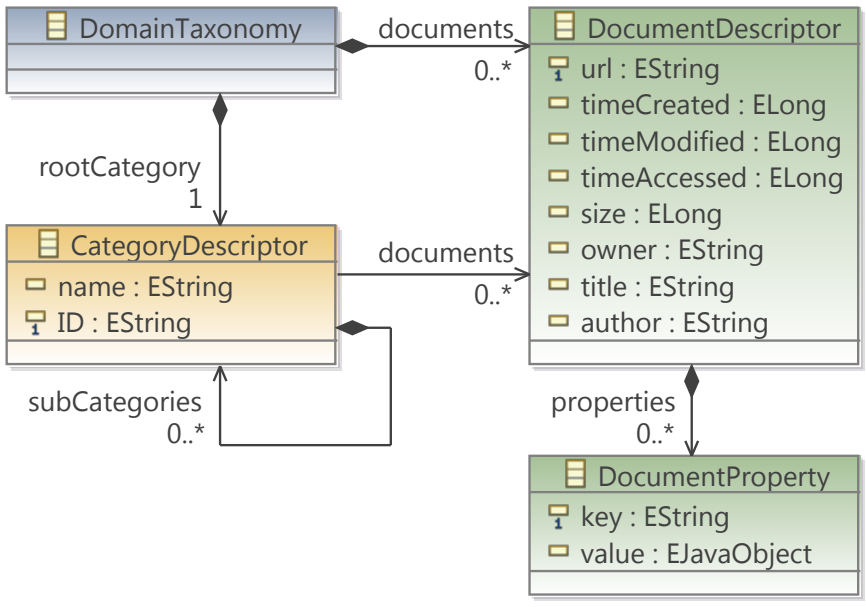


Abbildung 3.4: Übersicht über das Taxonomie-Modell

Das Wurzelelement wird im Modell (siehe Abbildung 3.4) durch einen speziellen *CategoryDescriptor* repräsentiert, welcher im Container *DomainTaxonomy* als *rootCatego-*

[7] Vergleiche Kapitel „2.1.2.1 Taxonomie“

*ry* referenziert wird. Ausgehend von diesem Wurzelelement können weitere Kategorien als Unterkategorien (*subCategories*) angelegt werden. Jede Kategorie wird durch einen Namen (*name*) und einen eindeutigen Identifikator (*ID*) beschrieben. Die Zuordnung eines oder mehrerer Dokumente im Sinne von *DocumentDescriptor*-Klassen (*documents*) zu einer Kategorie ist möglich, wobei nicht jede Kategorie Dokumente enthalten muss. Die Verwendung des Composite-Pattern ist eine ideale Variante zur Generierung einer Baumstruktur, wie sie für die Taxonomie benötigt wird.

Die Dokumente selber werden in der *DomainTaxonomy* verwaltet (*documents*) und können in mehreren verschiedenen Kategorien referenziert werden. Eine Mehrfachzuordnung eines Dokuments ist nicht ausgeschlossen. Jeder *DocumentDescriptor* beinhaltet Basisinformationen zu seinem abgebildeten Dokument. Dazu gehören die URL (*url*), die Zeitstempel (*timeCreated*, *timeModified*, *timeAccessed*), die Größe (*size*) und der Besitzer des Dokuments (*owner*). Zusätzlich werden zwei Attribute als feste Werte zu einem Dokument zugeordnet. Zum einen ist das der Dokumententitel (*title*) und zum anderen der Autor des Dokuments (*author*). Weitere Eigenschaften (*properties*) die von einem Dokument erfasst werden sollen, können über eine *DocumentProperty* realisiert werden. Eine *DocumentProperty* wird über ein Schlüssel-Werte-Paar charakterisiert.

#### 3.2.4.3 Verwendung des Modells

Das erstellte Taxonomie Modell wird in zwei Arbeitsschritten verwendet.

Im ersten Arbeitsschritt „**Erstellung von Templates**“ werden Taxonomie Templates mit Hilfe eines Editors (Taxonomie-Editor) erstellt. Es wird dabei ausschließlich der Kategoriebaum generiert und beschrieben. Die erstellten Templates werden anschließend für den zweiten Arbeitsschritt verwendet.

Im zweiten Schritt „**Annotation**“ werden Kopien der Templates erzeugt und befüllt. Die Einordnung von realen Dokumenten in die einzelnen Kategorien der Taxonomie erfolgt durch eine Import-Routine, in der die Dokumente auch zusätzlich beschrieben werden können. Eine anschließende Annotation der importierten Dokumente ist möglich.

### 3.2.5 Das Ontologie Modell

Für das vorliegende Projekt wurde eine modifizierte Variante des Topic Map Standard verwendet, um eine Ontologie[8] zu modellieren. Das erstellte Modell wird an den Gedanken und die Inhalte einer ISO Topic Map angelehnt, erhebt aber keinen Anspruch auf eine vollständige Umsetzung des ISO-Standards.

In diesem Abschnitt soll näher auf das erstellte Modell eingegangen werden. Dabei wer-

[8] Vergleiche Kapitel „2.1.2.3 Ontologie“

den die Kernelemente des Modells mit ihren Eigenschaften und Abhängigkeiten vorgestellt. Dazu zählen *Topic*, *TopicName*, *Association* und *TopicMap*. Zusätzlich wird am Ende des Kapitels auf die Bedeutung von *Scopes* eingegangen. Eine Übersicht über das komplette Modell ist im Anhang in der Abbildung A.1 zu finden.

#### 3.2.5.1 Begriffsreferenzen

Eine kurze Übersicht und Beschreibung der im Folgenden auftretenden Begrifflichkeiten bezüglich des Modells. [24]

**InformationItem** Das Interface *InformationItem* liefert einen eindeutigen Identifizierer (ID) zur einfachen Referenzierung der Elemente innerhalb der Topic Map.

**TopicMap** Die *TopicMap* ist das repräsentative Wurzelelement für das Modell und beinhaltet *Topic*- und *Association*-Elemente.

**Topic** Das *Topic* ist das primäre Konzept innerhalb der Topic Map. Es kann dabei zwischen einem Topic als Subjekt, einem Topic als Descriptor und einem Topic als Instanz unterschieden werden.

**TopicName** Jedes *Topic* kann mit Hilfe eines oder mehrerer Topic-Namen – *TopicName* – gelabelt werden. Die Vergabe von Namen an Topics ist essenziell für die Lesbarkeit und das Verständnis eines Topics. Ein Topic-Name besteht aus mehreren Einträgen.

**NameItem** Ein Topic-Name enthält eine Reihe von Namenseinträgen - *NameItem*. Diese dienen der Repräsentation von Namen für das entsprechende Topic.

**Facet** Das *Facet*-Element (Facette) besteht aus Key-Value-Paaren und ermöglicht das Hinzufügen von Eigenschaften zu Topics.

**FacetValue** Der *FacetValue* bildet den Wert der Facette und kann als Ziel ein Topic oder wieder ein *Facet* haben.

**Occurrence** Die *Occurrence* dient als Belegstelle für Topics. Ein Topic kann dabei keine, aber auch mehrere Occurrences besitzen.

**TopicType** Der *TopicType* bestimmt die Art eines Topics. Es dient wie auch das *InformationItem* mehr als interne Verwaltungshilfe.

**Scope** Mit Hilfe des *Scope* können Elemente in einen bestimmten Kontext eingeordnet werden. Er definiert Gültigkeitsbereiche innerhalb der Topic Map.

**Association** Die *Association* (Assoziation) verbindet zwei Topics miteinander und setzt sie in eine Beziehung.

**AssociationRole** Topics können innerhalb einer Assoziation verschiedene Rollen einnehmen, die sie näher klassifizieren.

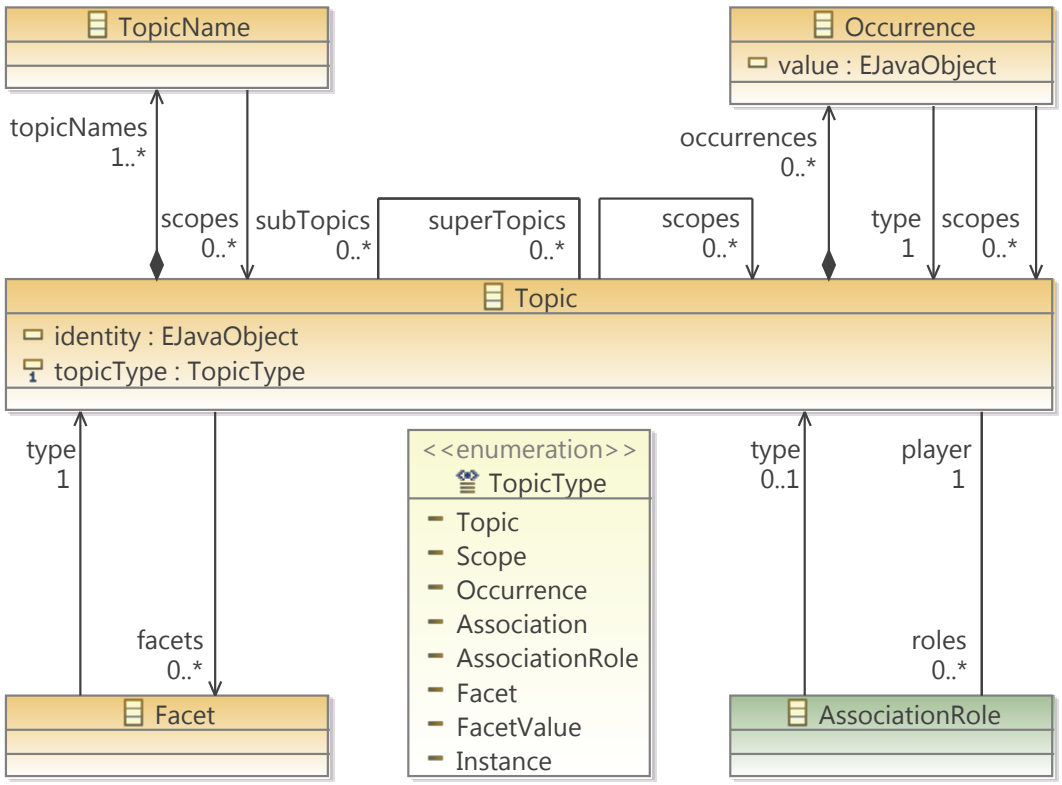


Abbildung 3.5: Modellierung des Topic-Elements

#### 3.2.5.2 Das Topic Element

Ein Topic repräsentiert ein eindeutig identifizierbares Subjekt und kann im Allgemeinen als die Verkörperung einer Sache angesehen werden – eine Person, ein Konzept, ein geschichtliches Ereignis, ein geografisches Objekt, eine Idee – ein Irgendwas, das existieren kann oder auch nicht. Im Folgendem wird nicht zwischen Topic und Subjekt unterschieden, sie werden als eine Einheit betrachtet. [9]

Das Topic ist der namensgebende Teil für die Topic Map, es beansprucht dementsprechend die größte Rolle im Modell. Es wird, wie auch alle anderen Elemente der Topic Map, über die Schnittstelle *InformationItem* eindeutig identifizierbar. Da diese interne Verwaltungshilfe keinerlei Auswirkungen auf den Grundgedanken des Modells hat, wird sie im Folgenden nicht weiter erwähnt. Die Identifizierung des Topics mit Hilfe einer Referenzangabe (*identity*) auf eine wie auch immer geartete Quelle, die das verkörperte Subjekt repräsentiert, ist optional möglich. Das Topic besitzt ein Attribut *topicType*, welche die Art des Topics näher beschreibt. Damit ist es möglich eine effiziente Zuordnung und Visualisierung des Topics bereit zu stellen. Der **TopicType** kann die folgende Werte

[9] Der ISO Standard unterscheidet genauer zwischen dem Subjekt und dem Topic. Das Subjekt ist eine Abstraktion, für die das Topic steht. Das Topic ist ein Objekt innerhalb der Topic Map, das dieses Subjekt repräsentiert: „The invisible heart of every topic link is the subject that its author had in mind when it was created. In some sense, a topic reifies a subject...“. [17]

annehmen:

- Topic — ein Topic im Sinne eines Subjekts (1)
- Scope — ein kontextgebendes bzw. einschränkendes Topic (2)
- Occurrence — ein Beschreibungselementen für Occurrences (2)
- Association — ein Typ für Assoziationsgruppen (2)
- AssociationRole — ein Typ für Assoziationsrollen (2)
- Facet — ein Schlüsselwert einer Eigenschaft (2)
- FacetValue — ein Eigenschaftswert (2)
- Instance — eine konkrete Ausprägung eines Topics (3)

Mit Hilfe dieser Einordnung der Topics ist auch die Ausprägung eines Topics als Subjekt (1), als Descriptor (2) oder als Instanz (3) erkennbar. Die Subjekt-Topics entsprechen den im einleitenden Absatz beschriebenen Topics als Verkörperung eines Subjekts. Die Descriptor-Topics beschreiben andere Elemente in der Topic Map bzw. dienen als beschreibende Elemente. Sie werden als Typen von anderen Elementen eingesetzt oder referenziert. Die Instanz-Topics sind Ausprägungen oder Instanziierungen von Subjekt-Topics. Diese Einteilung der Topics ermöglicht eine unabhängige Erstellung und Bearbeitung von Topic Maps - die Erstellung von Templates und die Benutzung bzw. die Instanziierung von Topic Maps. Durch den Grundsatz, dass jedes Element ein Topic oder vom Typ eines Topics ist, wird die Topic Map zu einer selbstbeschreibenden Informationsressource.

Topics können hierarchisch in einer Klassen-Instanz-Beziehung zueinander stehen. Die Oberklassen (*superTopics*) und Instanzen (*subTopics*) eines Topics werden je in einer Liste referenziert. Die Einordnung in eine Hierarchie ist dabei optional. Ein Topic kann verschiedene Charakteristika seiner Oberklasse übernehmen. So erbt ein Topic implizit die Beziehungen zu anderen Topics seiner abgeleiteten Oberklasse. [10]

Um ein Topic in einen bestimmten Kontext zu setzen, kann man eine Menge von Scopes (*scopes*) definieren. Die Bedeutung und Verwendung von Scopes (die auch an Zahlreichen anderen Stellen auftritt) wird in einem späterem Abschnitt behandelt.

Jedes Topic kann durch eine Liste von Eigenschaften *(factes)* näher beschrieben werden. Diese hält **Facet**-Elemente, welche über einen Typ (*type*) identifiziert werden und denen Werte (*values*) zugeordnet werden können (siehe Abbildung 3.6). Die Werte werden durch *FacetValue*-Elemente repräsentiert. Es muss mindestens ein Wert angegeben werden, es können aber auch mehrere Werte verwendet werden. Ein *FacetValue* besitzt genau ein *FacetTarget* (*value*), welches entweder ein Topic oder ein anderes *Facet*-Element ist. Auf diese Weise können verschachtelte Eigenschaften zu einem Topic

[10] Die Definition im ISO-Standard besagt, dass es sich klar um eine Klassen-Instanz-Beziehung handelt und nicht um eine Supertyp-Subtyp-Beziehung. Der Unterschied zwischen einer „ist-eine-Instanz-von“-Beziehung und einer „ist-eine“-Beziehung ist auf Modellebene durch die Vererbung von Eigenschaften interessant.

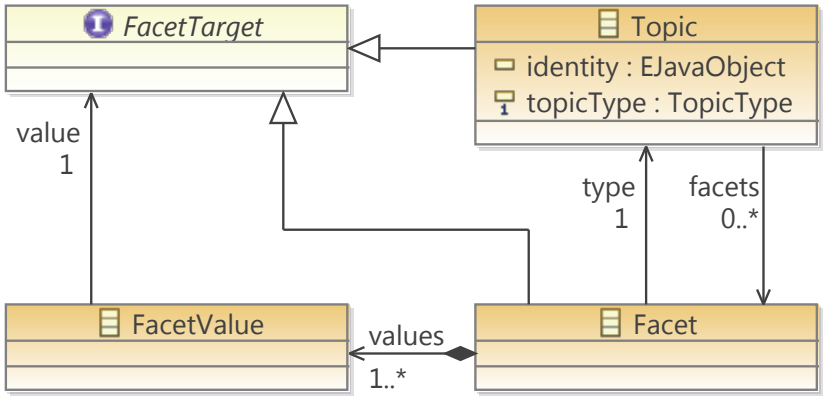


Abbildung 3.6: Modellierung des Facet-Elements

generiert werden. Der Facet-Typ, wie auch die dem *FacetValue* zugeordneten Topics als Werte, sind Descriptor-Topics.

Ein Topic kann mit verschiedenen Informationsquellen belegt werden, die in irgendeiner Hinsicht als relevant betrachtet werden. Solche Belegstellen werden als **Occurrences** (*occurrences*) bezeichnet. Normalerweise befinden sich diese Belegstellen (z.B.: Querverweise, Textbelege, Quellen, ...) außerhalb der Topic Map und es wird nur auf sie verwiesen (*value*). Verschiedene Occurrences von verschiedenen Topics können auch auf das selbe Objekt referenzieren und die Topics auf diese Weise in eine indirekte Beziehung setzen. Eine Occurrence ist von einem bestimmten Typ (*type*), welcher wiederum ein Topic ist, und kann mit Hilfe von Scopes (*scopes*) in einen Kontext eingeordnet werden.

Jedes Topic führt eine Liste von Rollen (*roles*), die es innerhalb einer Beziehung einnimmt. Die Verwendung von Associations wird in einem der nachfolgenden Abschnitte näher beschrieben.

Ein Topic besitzt einen oder mehrere zugeordnete *TopicName*-Elemente (*topicName*). Damit ist eine beschreibende Bezeichnung eines Topics möglich, um es benennen zu können – um es greifen zu können. Eine nähere Beschreibung der TopicName-Objekte wird im nächsten Abschnitt gegeben.

#### 3.2.5.3 Das TopicName Element

Ein Topic sollte beschreibbar sein, damit man die Möglichkeit hat über es zu sprechen. Im Modell wird jedem Topic eine Menge von *TopicName*-Elementen zugeordnet, die wiederum verschiedene Namenseinträge beinhalten. Ein *TopicName*-Element unterscheidet drei verschiedene Typen von Namenseinträgen:

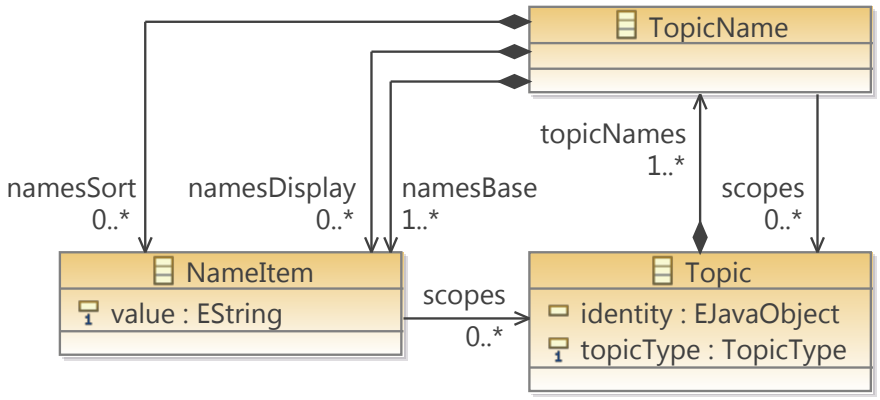


Abbildung 3.7: Modellierung des TopicName-Elements

- Standardbezeichnungen (*namesBase*)
- Anzeigenamen (*namesDisplay*) werden beim Visualisieren verwendet
- Sortierbegriffe (*namesSort*) werden bei Sortierverfahren verwendet

Es muss dabei mindestens ein Standardbezeichner für jeden *TopicName* vergeben werden. Existiert kein Anzeigename oder Sortierbegriff, so wird stattdessen der Standardbezeichner verwendet. Eine erweiterte Referenzierung der Anzeigenamen kann benutzt werden um beschreibende Elemente oder Objekte für eine grafische Visualisierung bereit zu stellen (z.B.: Anweisungen wie das Objekt in einer grafischen Umgebung gezeichnet werden soll).

Jeder Namenseintrag-Typ wird intern als eine Liste von **NameItem**-Elementen verwaltet. Ein *NameItem* besitzt einen Wert (*value*), welcher den Namen repräsentiert, sowie eine Liste mit Gültigkeitsbereichen (*scopes*).

Es ist Möglich einem Topic mehrere *TopicName*-Elemente zuzuordnen. Jedem *TopicName* kann dabei ein separater Kontext (*scopes*) zugewiesen werden, in dem er gültig ist. Die Erstellung von verschiedenen Sprachräumen innerhalb einer Topic Map könnte auf diese Weise realisiert werden.

Die Vergabe von zwei identischen Namen innerhalb eines Gültigkeitsbereiches ist nicht erlaubt. Topics mit gleichen Namen werden zu einem Topic zusammengefasst, welches die Charakteristiken beider Topics zusammenführt. [11] Ebenfalls zu einer Zusammenführung mehrerer Topics führt das Auftreten von zwei gleichen Identitäten (*identity*).

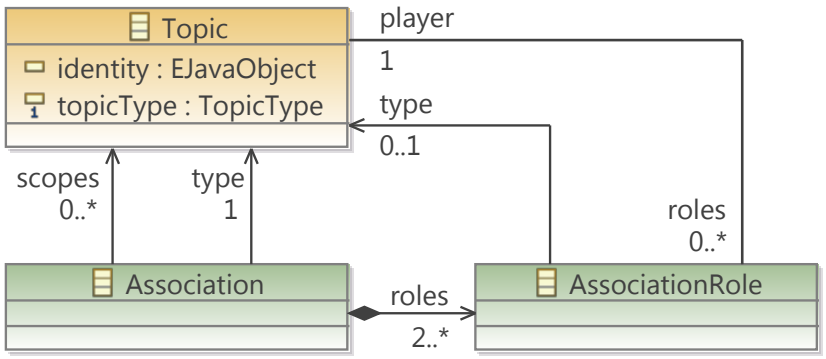


Abbildung 3.8: Modellierung des Association-Elements

#### 3.2.5.4 Das Association Element

Die Association ermöglicht es Beziehungen zwischen Topics zu beschreiben. Jede Association ist von einem bestimmten Typ (*type*) in Form eines Topics und wird in einen Kontext (*scopes*) gesetzt.

Die Modellierung von Eigenschaften der Beziehung, wie zum Beispiel die Symmetrie oder die Transitivität können über die Eigenschaften des Assoziationstyps erfolgen. Es ist damit nicht notwendig weitere Attribute für eine Association festzulegen.

In einer Beziehung müssen mindestens zwei Rollen (*roles*) definiert sein, die vom Typ **AssociationRole** sind. Eine Rolle kann von einem bestimmten Typ (*type*) sein, welcher als Topic referenziert wird, oder ohne spezifizierten Typ. Falls die Rolle keinen Typ besitzt, so stellt sie das entsprechende Ziel-Topic (*player*) nur innerhalb der Beziehung dar, ohne näher auf den Status einzugehen. Die Anzahl der Rollen ist nach oben hin nicht limitiert, um die Vergabe von mehreren Rollen für ein Topic (*player*) zu ermöglichen. Es wird nicht verboten, mehr als zwei Topics über eine *Association* in Beziehung zu setzen. Es wird aber empfohlen nur zwei Topics in eine Beziehung zu setzen. Für die Modellierung mehrerer Teilnehmer sollten jeweils eigene *Association*-Elemente erstellt werden. Eine Selbstreferenzierung, um Beziehungen eines Topics zu sich selbst zu definieren, ist ebenfalls nicht ausgeschlossen.

Einige Eigenschaften der Topic Map Elemente, wie zum Beispiel die Klassen-Instanz-Beziehung der Topics, könnten über Associations modelliert werden. Diese Eigenschaften sind aber so allgemeingültig und universell, dass es sinnvoll ist, diese zu Standardisieren um die Interoperabilität zwischen verschiedenen Topic Map System zu optimieren.

[11] Es wird dabei von dem „Topic naming constrain“ gesprochen.

#### 3.2.5.5 Das TopicMap Element

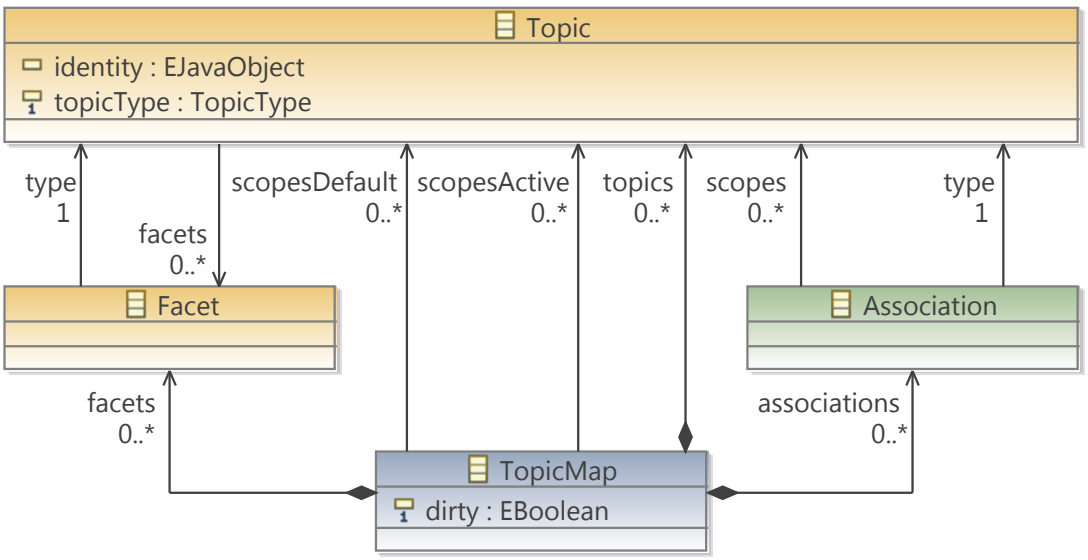


Abbildung 3.9: Modellierung des TopicMap-Elements

Das TopicMap Objekt bildet das Wurzelelement für das Modell. Es implementiert das Interface *InformationItem* um im System mittels einer *ID* eindeutig identifiziert werden zu können. Die Identifizierung spielt vor allem beim Einsatz von Datenbanken zur Persistierung eine wichtige Rolle.

Das *TopicMap*-Objekt verwaltet alle Assoziationen (*associations*) und Topics (*topics*) in je einer Liste. Es beinhaltet eine Liste mit allgemeingültigen Scopes (*defaultScopes*), die für diese Topic Map und alle darin auftretenden Elemente indirekt verwendet werden, sowie eine Liste mit aktuell aktiven Scopes (*scopesActive*), die auf die Topic Map und alle darin auftretenden Elemente direkt angewendet werden.

Diese indirekten Scopes sind die Rahmenbedingungen für die Topic Map und werden allen Topics zugeordnet, sollte die Topic Map mit einer andern Topic Map zusammengeführt werden. Die aktiven Scopes dienen einer einfachen internen Verwaltung und Prüfung von gültigen Scopes, sie beschreiben die vom Nutzer als derzeit vorherrschende Gültigkeitsbereiche für die Topic Map.

Als zusätzliches Attribut, besitzt das Objekt ein boolesches *dirty*-Flag, welches der Regulierung der Persistierung dient. Es wird bei einer Modifizierung der Topic Map umgesetzt. Dieses Attribut ermöglicht eine Auslösung der Persistenz-Routine, ohne einen kompletten Abgleich aller Topic Map Elemente durchführen zu müssen.

#### 3.2.5.6 Verwendung von Scopes

Topics können eine Menge an Charakteristika aufweisen (Namen, Belegstellen, Rollen, ...). Die Zuweisung einer Charakteristik zu einem Topic wird unter bestimmten Richtlinien, ungeachtet ob diese explizit spezifiziert werden, als gültig angenommen. Dieser Gültigkeitsbereich oder Kontext wird als Scope bezeichnet. Ein Scope ist wiederum die spezielle Ausprägung eines Topics.

Durch den Einsatz von Scopes können explizit Gültigkeitsbereiche für Topics, TopicNames, NameItems, Occurrences, Associations und die TopicMap an sich erstellt werden. Besitzt ein Element keinen zugewiesenen Scope, dann wird es als uneingeschränkt behandelt und ist immer gültig. [12] Die Ausnahme ist, wenn ein übergeordnetes Element einen Scope besitzt. Dieser wird implizit von Oben nach Unten weitergereicht und schränkt die Gültigkeit ein. Die Abhängigkeiten der Scopes sind in der folgenden Tabelle aufgeführt:

| **Ausgangselement** | **steht in Abhängigkeit zu** |
|---|---|
| NameItem | TopicName |
| TopicName | Topic (zugehöriges Topic) |
| Association | Topic (Topic Typ)<br>Topic (Teilnehmende Topics (AssociationRole)) |
| Occurrence | Topic (zugehöriges Topic)<br>Topic (Topic Typ) |
| Topic | TopicMap |

Ein Element ist gültig, wenn es entweder uneingeschränkt ist oder den aktiven Scope komplett enthält. Die Gültigkeit von Elementen kann anhand der nachfolgenden Tabelle verdeutlicht werden. Die Kopfzeile stellt den gerade angewendeten Scope dar, die Zeilen verschiedene Elemente mit einem entsprechenden Scope. Ein „✓“ steht für ein gültiges Element, „∅“ steht für „kein Scope“.

| | ∅ | A | A, B | A, B, D |
|---|---|---|---|---|
| ∅ | ✓ | ✓ | ✓ | ✓ |
| A | ✓ | ✓ | | |
| B | ✓ | | | |
| C | ✓ | | | |
| A, B | ✓ | ✓ | ✓ | |
| A, B, C | ✓ | ✓ | ✓ | |

[12] Im ISO-Standard wird vom „unconstrained scope“ gesprochen.

Es entsteht die Möglichkeit verschiedene Sichten auf die Topic Map zu erstellen ohne die eigentlichen Inhalte modifizieren zu müssen. [13] Ebenso ist die semantische Fehlerquote beim Zusammenführen von mehreren Topic Maps dadurch nahezu auf Null reduzierbar.

#### 3.2.5.7 Verwendung des Modells

Das erstellte Topic Map Modell wird in drei Arbeitsschritten verwendet.

Im ersten Arbeitsschritt „**Erstellung von Templates**“ werden Topic Map Templates mit Hilfe eines Editors (Topic Map-Editor) erstellt. Dabei ist es notwendig die Subjekt-Topics und Descriptor-Topics, sowie Beziehung zwischen diesen zu definieren. Weiterhin können Eigenschaften und Scopes festgelegt werden. Die erstellten Templates werden anschließend für den zweiten Arbeitsschritt bereitgestellt.

Im zweiten Schritt „**Annotation**“ werden Kopien der Templates erzeugt und befüllt. Eine Instanziierung der Templates zu projektbezogenen Topic Maps erfolgt durch Anreicherung durch Instanz-Topics, neue dadurch entstehende Beziehungen, Eigenschaften, Occurrences und Scopes für spezielle Einsatzbereiche. Dieser Schritt wird während einer Annotationsphase von Dokumenten durchgeführt.

Ist eine projektbezogene Topic Map einsatzbereit, können im dritten Schritt „**Abfragen von Topic Maps**“ Anfragen an diese gestellt werden. Über eine entsprechende Oberfläche wird die Topic Map durchsucht und zur Analyse der eingespeisten Informationen verwendet.

Projektbezogen werden Scopes primär für die Multilingualität verwendet und schränken dadurch die Informationsressourcen nicht ein, sondern helfen bei der Bedeutungsfindung. Dabei handelt es sich nicht nur um Muttersprache und Fremdsprachen, es sollen über dieses System vorrangig Dialekte und Slang-behaftete Sprachräume abgedeckt werden. Die Verwendung von vorhandenen Thesauri[14] zur Erweiterung des Suchraumes für Begriffe, über das Heranziehen von Synonymen, kann eine Unterstützung bei der Modellierung und bei der Abfrage der Topic Map darstellen.

#### 3.2.5.8 Beispiel

Dieser beispielhafte Auszug aus dem Topic Map Modell zeigt die Zusammenarbeit der verschiedenen Modellelemente. Mittig Links in der Abbildung ist ein roter Kreis zu sehen (1) - dieser stellt ein Subject mit einem TopicName dar, welcher das NameItems

[13] Die Mehrdeutigkeit von Homonymen kann damit eliminiert werden. Ein Blatt, das an einem Baum hängt, kann von einem Blatt in einem Buch separiert werden.

[14] Vergleiche Kapitel „2.1.2.2 Thesaurus“

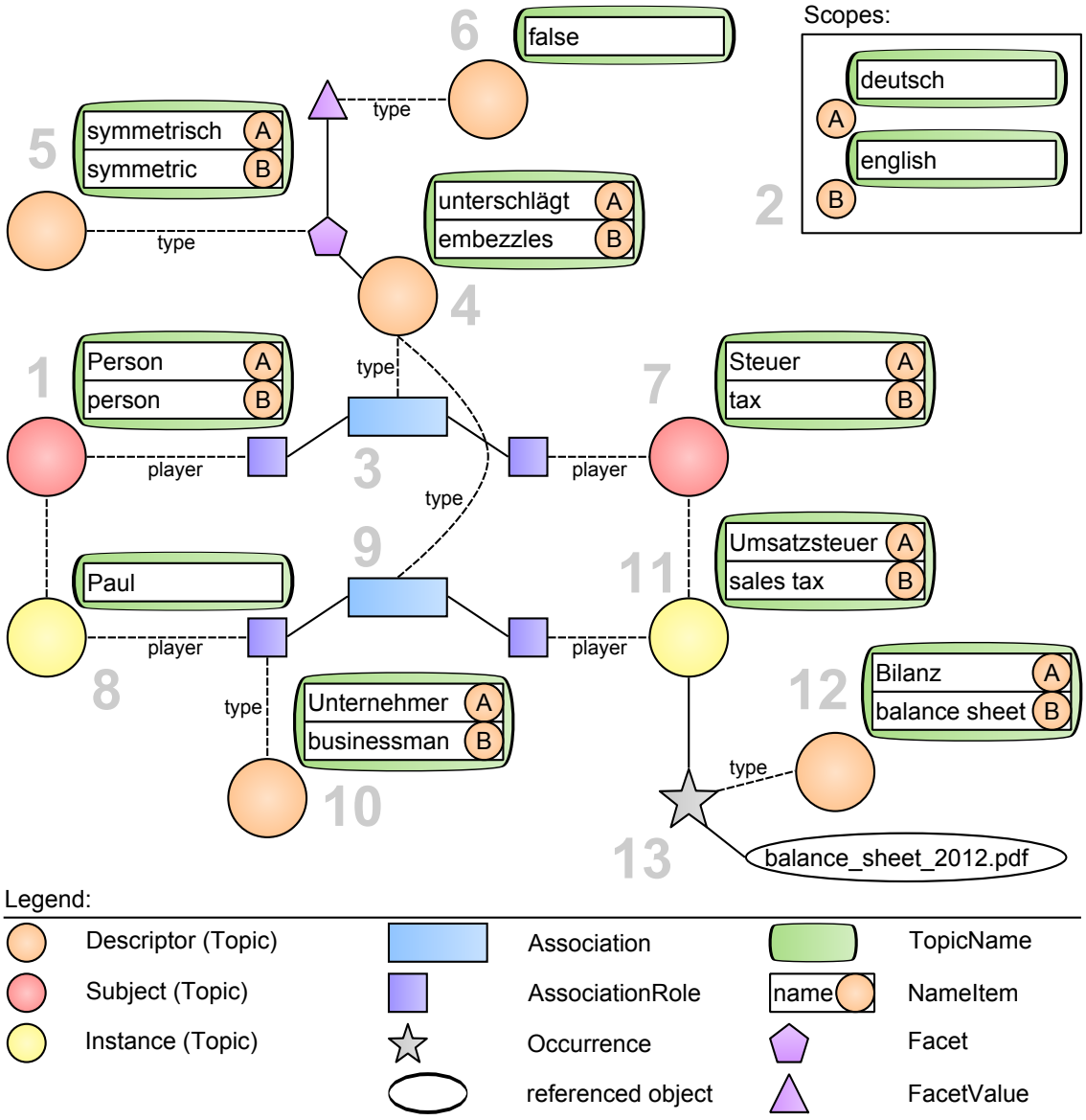


Abbildung 3.10: Beispielauszug aus dem Topic Map-Modell [24]

*Person* in einem Scope A und das NameItem *person* in einem Scope B beinhaltet. Die entsprechenden Scopes, die auch an anderen Stellen verwendet werden, können in der oben rechten Ecke der Abbildung gefunden werden (2). Diese Desciptoren mit einem einzelnen uneingeschränktem NameItem beschreiben die Gültigkeit von anderen NameItems innerhalb der Topic Map. Wird beispielsweise Scope A angewendet, so werden alle NameItems deaktiviert, die nicht diesen Scope A besitzen. Mit einem aktivem Scope A (*deutsch*), wird das Subject als *Person* angezeigt. Dieses Subject ist über eine unspezifizierte AssociationRole mit einer Association (3) verbunden. Die Association ist vom Typ *unterschlägt*, welcher durch einen Descriptor (4) repräsentiert wird. Dieser Descriptor besitzt ein Facet vom Typ *symmetrisch* (5) und beinhaltet ein FacetValue *false* (6) - beide Werte werden ebenfalls durch Descriptoren beschrieben. Der zweite Teilnehmer *Steuer* (7) der Association *unterschlägt* ist durch eine weitere AssociationRole verbunden. Auf diese Weise ist die asymmetrische Relation “Person unterschlägt Steu-

er" entstanden. In der unteren linken Hälfte der Abbildung besitzt die Instance *Paul* (8) als eine konkrete Ableitung des Subjects *Person* ebenfalls eine Association *unterschlägt* (9). Anders als zuvor wird die Instance über eine spezifizierte AssociationRole vom Typ *Unternehmer* (10) klassifiziert, welche die Rolle von *Paul* innerhalb der Relation festlegt. Der andere Teilnehmer dieser Relation ist ebenfalls eine Instance. Die Instance *Umsatzsteuer* (11) vom Typ *Steuer* wird über eine Occurrence vom Typ *Bilanz* (12) mit Hilfe des referenzierten Wertes *balance_sheet_2012.pdf* (13) belegt. Resultierend aus dieser kleinen Beispielmodellierung kann der Fakt, dass die Person Paul als Unternehmer Umsatzsteuern unterschlagen hat, welche in der Datei "balance_sheet_2012.pdf" auftreten, extrahiert werden.

### 3.2.6 Annotation

Die Annotation von Dokumenten erfolgt durch die Kombination der beiden Modelle, Taxonomie und Topic Map, wie in Abbildung 3.11 gezeigt wird.

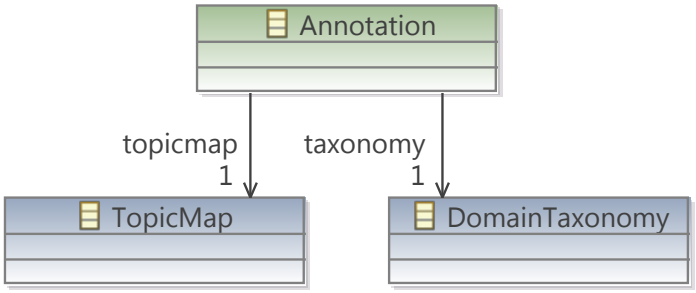


Abbildung 3.11: Übersicht über des Annotations-Modell

Die Wurzelelemente der jeweiligen Modelle werden als *taxonomy* und *topicmap* referenziert und dem Meta-Modell als Abhängigkeiten bekannt gemacht. EMF ist in der Lage die Modell-Ressourcen auch bei Änderungen aktuell zu halten, indem es jegliche Funktionalität an die Adapter und Provider der referenzierten Modelle delegiert. Die Modell-Plugins müssen dafür zur Verfügung stehen und geladen sein.

Das Modell bringt keine neuen Modell-Elemente oder Funktionalitäten in das System ein. Die Annotation von Dokumenten erfolgt durch die Einordnung eines *DocumentDescriptor*-Objekts innerhalb der Taxonomie und eine anschließende Instanziierung von *Topic*-Objekten vom Typ „Instance", sowie die Festlegung aller entsprechender Eigenschaften und Attribute.

### 3.2.7 Benutzer

Das Benutzer-Modell, in Abbildung 3.12 in der Übersicht dargestellt, dient der Klassifizierung von Benutzern.

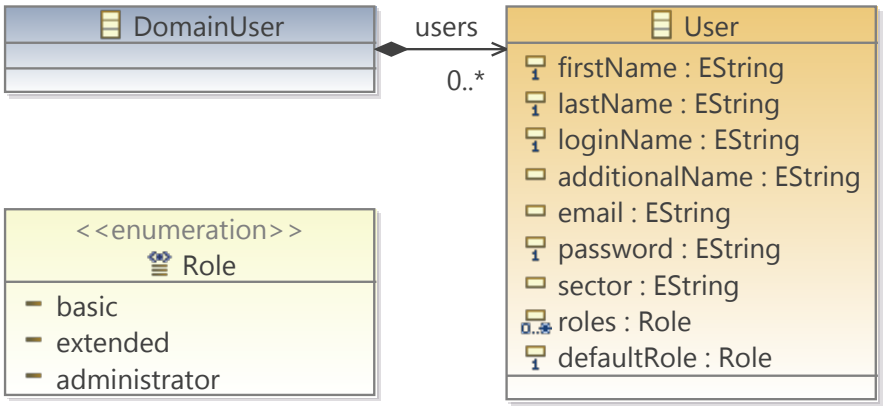


Abbildung 3.12: Übersicht über des Benutzer-Modell

Es beinhaltet Attribute zur eindeutigen Identifizierung von Benutzern, wie den Vorname (*firstName*), den Nachname (*lastName*) und einen wählbaren Login-Name (*loginName*). Die Verwendung eines Passworts (*password*) in Kombination mit dem Login-Namen ermöglicht die Zugriffssteuerung auf das System für einen Benutzer. Namenszusätze (*additionalName*) und die Zuordnung eines Arbeitsbereichs (*sector*) können als optionale Angaben vergeben werden. Soll eine Benachrichtigung eines Nutzers automatisch durch das System erfolgen können, so ist die Angabe einer eMail-Adresse (*email*) möglich. Die entscheidenden Bestandteile in der Modellierung ist die Vergabe von vordefinierten Rollen (*roles*) an einen Benutzer, die als Administrator (*administrator*), Erweitert (*extended*), Basis (*basic*) oder einer Kombination dieser Rollen möglich ist. Zusätzlich wird eine Rolle als Standardrolle festgelegt, die bei der Anmeldung am System als primäre Rolle verwendet wird.

Die Rollen eines Benutzers bzw. das Benutzer-Modell als Ganzes liegen in keiner Relation zu den anderen erstellten Modellen.

**Verwendung des Modells**

Die Anmeldung am System mit Hilfe eines Benutzerprofils weist verschiedene Vorzüge auf.

Über ein Logging-System kann jederzeit nachvollzogen werden, welcher Benutzer zu welcher Zeit am System angemeldet war und welche Aktionen er durchgeführt hat. Es kann beispielsweise ein Protokoll erstellt werden, welche Schritte durchgeführt wurden um ein bestimmtes Ziel zu erreichen. Weiterhin ist es möglich benutzerspezifische Einstellungen, wie das Layout der Anwendung, geöffnete Editoren und Ressourcen zu speichern und beim Neustart wieder zu laden. Es ist Unbefugten nicht möglich unbeaufsichtigt in das System einzudringen. Die Verwaltung von Restriktionen bezüglich erlaubter Operationen kann durch das Laden von Plugins je nach Nutzerrollen durchgeführt werden.

## 3.3 Systemgrundlagen

Die Betrachtung der Systemarchitektur erfolgt von Außen nach Innen. Es werden zuerst Rahmenkonzeptionen vorgestellt und danach werden die einzelnen Komponenten vertieft.

### 3.3.1 Eclipse

Die Modell- und Systemgrundlagen werden stark durch die zugrunde liegende Entwicklungsumgebung beeinflusst. Das OSGi-Framework, als bekanntester Standard zur Software-Modularisierung im Bereich Java, bildet seit der Version 3.0, mit seiner Referenzimplementierung Equinox, die Basis für die Eclipse IDE. Eclipse selbst ist ein hochgradig modulares Framework, das auf Grund der starken Community, die es entwickelt und verwendet, neue Technologien innerhalb kürzester Zeit integrieren kann. Die Möglichkeit der Verwendung von Java und die daraus resultierende Plattformunabhängigkeit, die starke Modularisierung auf Grundlage von OSGi und der Eclipse Plattform selbst, sowie die aktive Community, die das Projekt vorantreibt, sind optimale Grundlagen für die gestellten Eigenschaften an das Zielsystem und der Grund für die Verwendung von Eclipse für dieses Projekt.

In dieser Arbeit wurde die Eclipse IDE in der Version 3.7 (Eclipse Indigo) verwendet. Zum Start der Entwicklung des Projekts ist das Programmiermodell der Eclipse IDE 3.x wesentlich stabiler und kann ein größeres Spektrum an Tools aufbringen als das neue Programmiermodell der Eclipse 4.x IDE – wobei die 4.x Version wesentlich flexibler einsetzbar und einfacher in der Anwendung ist. [34] Die Entscheidung der Verwendung von Eclipse 3.x gegenüber Eclipse 4.x wurde vorrangig auf der Grundlage der höheren Stabilität und besseren Dokumentation getroffen, welches auf das Fehlen eines stabilen Release der Version 4.x zurückzuführen ist.

Bei der Entwicklung von Eclipse-Anwendungen gibt es Richtlinien, deren Einhaltung und Umsetzung die Anwendungsentwicklung hinsichtlich Erweiterbarkeit, Weiterentwicklung und Strukturierung stark fördern. Die Hausregeln für die Erstellung von Eclipse-Anwendungen (siehe Anhang B) wurden bei der Konzeption und Implementierung des erstellten Systems im Auge behalten.

### 3.3.2 Framework

Frameworks werden im Allgemeinen unter dem Gesichtspunkt einer Wiederverwendung von architektonischen Mustern entwickelt und genutzt. Solche Muster sind nicht ohne eine konkrete Anwendungsdomäne im Hintergrund definierbar. Aus diesem Grund sind Frameworks meist domänenspezifisch oder auf einen bestimmten Anwendungstyp beschränkt. Ein Domain-Framework stellt ein Programmiergerüst für einen bestimmten

Problembereich zur Verfügung, in dem eine Anwendung erstellt werden kann, die zur Lösung dieses Problembereichs typischerweise benötigt wird. Genauer betrachtet wird keine fertige Anwendung erstellt, es werden Funktionen, Strukturen und Entwurfsmuster für eine Anwendung bereitgestellt. Das Framework gibt somit die Anwendungsarchitektur vor. Die Architektur kann über Schnittstellen erweitert werden, wobei Module (oder auch Plugins) registriert werden, welche dem Framework ihre Funktionalität anbieten, wobei das Framework an sich nicht modifiziert wird. Es definiert den Kontrollfluss der Anwendung und greift ausschließlich auf die angebotenen Funktionalitäten zurück. [4, Seite 25f.]

### 3.3.3 Rich Client Platform (RCP)

Die Entwicklungsumgebung Eclipse unterstützt ab der Version 3.0 die Wiederverwendung der Eclipse Plattform um eigenständige Anwendungen zu entwickeln, die die gleiche Technologie verwenden, wie die Eclipse IDE. [34]

Eine *Rich Client Platform (RCP)* ist ein allgemeiner Anwendungsrahmen[15], der im Kern aus einer geringen Anzahl an Modulen und Paketen besteht, welche einen Standardanwendungsrahmen mit Perspektiven, Sichten und Editoren, ohne irgendwelche anderen Tools oder IDE-spezifischen Aspekte (Quellcode-Bearbeitung, Refactoring, Kompilierung, ...) bereitstellt. Es stellt damit die Infrastruktur für eine Anwendung, ohne die Notwendigkeit allgemeine oder spezifische Tools zu integrieren. [3, Seite 817]

Die RCP verwendet systemeigene Oberflächenkomponente, welche stabil, verlässlich und schnell sind. Der stark modulare Ansatz der Plattform ermöglicht es Komponentenbasierte Systeme zu entwickeln. Die Minimalausstattung für eine RCP-Anwendung (mit Oberfläche) ist die Hauptanwendung (äquivalent zur main-Methode der Standard-Java Anwendungen), das Layout (oder die Perspektive) und ein sogenannter „Workbench Advisor“, welcher als unsichtbare technische Komponente das Aussehen der Oberflächenelemente kontrolliert. [34] Diese Komponente werden im Kapitel 4.1.1 eingehender behandelt.

### 3.3.4 Plugin

Die Eclipse Plugin Struktur besteht aus einem kleinen Kern mit einem Plugin-Loader, welcher von verschiedensten Plugins umgeben ist. Der Kern selbst ist eine Implementierung nach der OSGi[16] R4 Spezifikation und stellt die Umgebung, in der Plugins ausgeführt werden können. Der Plugin-Loader scannt alle Plugins zum Anwendungsstart und ordnet diese intern anhand der Abhängigkeiten zueinander an ohne die Plugins

[15] „... a generic application framework called Rich Client Platfform, or simply RCP.“ [3]

komplett zu laden. [3, Seite 107]

In Eclipse wird eine Software-Komponente als Plugin bezeichnet. Jedes Plugin besitzt eine Datei „META-INF/MANIFEST.MF“ und zusätzlich eine „plugin.xml“ Datei. Diese beiden Dateien werden zusammen als das Plugin Manifest bezeichnet und definieren allgemeine Informationen und Abhängigkeiten des Plugins, sowie die Interaktion des Plugins mit anderen Komponenten des Systems. Die Manifest Struktur erleichtert das sogenannte „lazy-loading“, bei dem ein Plugin erst vollständig geladen wird, wenn sein Code tatsächlich gebraucht wird. Damit kann die Ladezeit der Anwendung, sowie auch der Speicherbedarf reduziert werden. Der Plugin-Loader benötigt nur die Plugin-Manifeste um die zum Anwendungsstart generierte Struktur zu erstellen. [3, Seite 108]

Optional kann einem Plugin eine Klasse angefügt werden, welche das Plugin von einem programmatischen Standpunkt aus repräsentiert. Diese Datei wird als „Activator“ bezeichnet und kontrolliert den Lebenszyklus des Plugins. Der *Activator* wird über die gesamte Lebenszeit des Plugins verwendet – es werden keine anderen Instanzen von ihm erzeugt. Über den *Activator* können statische Ressourcen für das Plugin bereit gestellt werden, sowie Statusinformationen zwischen verschiedenen Session über die `start()` und `stop()` Methoden gesichert bzw. geladen werden. [3, Seite 120]

### 3.3.5 Extension Points

Extension Points (engl. Erweiterungspunkte) definieren Schnittstellen für andere Plugins, damit diese ihre eigene Funktionalität einbringen können. Die Extension Points sind in einer speziellen Verwaltungsdatei (*plugin.xml*) abgelegt und werden während des Starts der Anwendung mitsamt Abhängigkeiten erfasst. [34] Es existieren zwei Richtungen in die Erweiterungen vorgenommen werden können. Zum einen können Plugins selber Extension Points definieren um die eigene Funktionalität erweitern zu lassen. Das Plugin muss dabei die entsprechenden Beiträge zu dieser Erweiterung überwachen und evaluieren. Zum anderen kann ein Plugin eine Erweiterung zu einem anderen Plugin bereit stellen. [17] Es muss dafür einen existierenden Extension Point adressieren und entsprechende Daten oder entsprechenden Quellcode anbieten. [31] Ein Extension Point wird mit Hilfe einer eindeutigen ID angesprochen (z.B.: „org.eclipse.core.runtime.application“).

Sie ermöglichen die deklarative Beschreibung von Komponenten und Funktionalitäten, welche einfach zur Erweiterung von Plugins durch andere Plugins verwendet werden

[16] Im konkreten wird „Equinox“, eine Referenzimplementierung des OSGi Frameworks, verwendet, welches die stärken der OSGi Spezifikation im Rahmen von einem stabilen Komponentenmodell und der Unterstützung von dynamischem Verhalten umsetzt.

[17] Eine Referenz-Übersicht über alle derzeit unterstützten Extension Points: `http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/extension-points/index.html`

können. Diese hohe Modularität und Funktionalität legt einen ausgezeichneten Grundstein für ein hochgradig anpassbares Framework.

### 3.3.6 Perspectives

Eine Perspective (Perspektive) ist eine Möglichkeit die Oberflächenelemente für eine bestimmte Aufgabe auszurichten. Das generelle Vorgehen für das Erstellen eines Layouts von Eclipse RCP-Anwendungen besteht darin, Sichten (Views) um einen festen Editorbereich anzuordnen. Der Editorbereich ist dabei das zentrale Element und kann nicht entfernt werden. [3, Seite 5f.]

Perspektiven werden über den Extension Point „org.eclipse.ui.perspectives“ verfügbar gemacht und definieren das Layout der Oberfläche über eine Klasse mit der Schnittstelle „IPerspectiveFactory“. Das Interface stellt die Methode `createInitialLayout()` bereit, über die ein Anfangslayout generiert werden kann. Es können feste Views eingebunden oder Platzhalter definiert werden, an denen bestimmte Views eingebunden werden sollen, sobald diese aufgerufen werden. [3, Seite 423ff.]

Für das Projekt wird eine einzelne Perspektive erstellt, welche linksseitig eine Möglichkeit für die Navigation registriert und den restlichen Bereich für die Editoren verfügbar lässt. Es werden vorerst keine weiteren Views benötigt oder bereit gestellt.

### 3.3.7 Views und Editoren

In der Eclipse Welt existieren zwei grundsätzliche Elemente zur Anzeige und Bearbeitung von Ressourcen, zum Einen sind das Editoren und zum Anderen Sichten (Views). Views und Editoren haben viele gemeinsame Eigenschaften im Verhalten durch die implizierte Oberklasse „org.eclipse.ui.WorkbenchPart“, sowie das Interface „org.eclipse.ui.IWorkbenchPart“. [3, Seite 354]

Die Erzeugung eines Views und eines Editors erfolgt ebenfalls in ähnlichen Schritten. Es muss eine Extension im Manifest registriert werden und eine Klasse mit dem eigentlichen Quellcode muss erstellt werden. Für einen View kommt noch die Zuweisung zu einer Kategorie hinzu, um die Übersicht bei der großen Menge an möglichen Views nicht zu verlieren. Beide Elemente können mit jeglichen Komponenten bestückt werden, wobei es üblich ist einen View möglichst schlicht zu halten, mit nur einer Komponente. [3, Seite 291ff.]

Der wichtigste Unterschied liegt in der Vorgehensweise des Ressourcen-Managements. Ein View wendet eine ausgeführte Aktion sofort auf eine entsprechende Ressource (oder die Arbeitsfläche) an, wobei Editoren dem klassischen Paradigma von Öffnen-Bearbeiten-Speichern folgen. Weiterhin werden Editoren ausschließlich im Editorbe-

reich angezeigt. Views hingegen können frei um diesen Bereich angeordnet werden. Ein View zeigt Informationen zu einer oder mehreren Ressourcen an oder etwas komplett Ressourcenunabhängiges (z.B.: Netzwerkstatus, verfügbarer Speicher, ...). Editoren auf der anderen Seite sind ausschließlich Ressourcen-basiert und können nur in Verbindung mit einer Ressource ausgeführt werden. [3, Seite 291ff.]

### 3.3.8 Widgets

Die grafischen Oberflächenkomponenten werden in Java als Widgets bezeichnet. Moderne Java-Anwendungen stützen sich dabei hauptsächlich auf zwei Toolkits, die aufeinander aufbauen.

Standard Widgets wie Label, Textboxen, Buttons, etc. werden durch die Verwendung von SWT unterstützt. Das Standard Widget Toolkit bildet eine dünne Schicht über den systemeigenen Oberflächenkomponenten, und bildet die entsprechenden Widgets direkt auf native Komponente via JNI ab oder emuliert diese selber, falls sie nicht von der Plattform unterstützt werden. Es ist beschränkt auf einfache Datentypen (Text, Zahlen, Bilder), was eine umständliche Handhabung von objektorientierten Daten, welche in Listen, Tabellen, Bäumen oder anderen Konstrukten dargestellt werden müssen, zur Folge hat. [3, Seite 135ff.] [34]

Für objektorientierte, komplexere Daten bieten sich die Verwendung von JFace an, welches auf SWT aufsetzt. Einer der interessanten Features ist der JFace Viewer. Viewer sind modellbasierte Adapter für verschiedene SWT Widgets, welcher die Darstellung von Daten als Listen, Tabellen oder Bäumen stark vereinfacht. Der Vorteil liegt in der Benutzbarkeit von Modellen ohne das diese manuell in ihre Basiselemente zerlegt werden müssen. Realisert wird dies über die Verwendung von Adapter-Schnittstellen um beispielsweise den Namen eines Elements (*LabelProvider*) oder dessen Kind-Elemente (*ContentProvider*) zu extrahieren oder Elemente nach bestimmten Eigenschaften zu sortieren und zu filtern. Ein *ContentProvider* nimmt das Modell entgegen und zerlegt es in relevante Elemente, die der Nutzer visualisieren möchte. Der *LabelProvider* verwendet diese Elemente und kann ihnen einen Anzeigetext (ein Label) und ein Icon zuweisen. Jeder Viewer benötigt genau einen *ContentProvider* zur Aufbereitung des Modells und kann je nach Viewer-Typ mehrere *LabelProvider* unterstützen (Tabellenspalten können jeweils einen eigenen *LabelProvider* verwenden). [3, Seite 193ff.] [34]

### 3.3.9 Dialoge und Wizards

Ein Dialog stellt ein Oberflächenelement dar, welches über allen anderen Oberflächenelementen innerhalb einer Anwendung steht und diese so lange blockiert, bis die angeforderten Informationen eingegeben wurden. Plattformspezifische Dialoge ermöglichen den Zugriff auf Systemressourcen, wie zum Beispiel das Dateisystem, ohne großen Auf-

wand, können dabei aber nur geringfügig in Layout und Funktionalität modifiziert werden. Ein Wizard ist eine besondere Subklasse des Dialogs und kann mehrere Seiten beinhalten um eine Reihe von Informationen übersichtlich abzufragen. Dialoge werden üblicherweise über die integrierten SWT-Bibliotheken realisiert, während Wizards mittels der JFace-Bibliothek plattformunabhängig generiert werden. Das ermöglicht eine freiere Gestaltung der Wizards gegenüber der Standard-Dialoge auf Kosten von Mehraufwand bei der Implementierung, wobei natürlich auch benutzerdefinierte Dialoge generiert werden können. [3, Seite 441ff.]

Die im Projekt verwendeten benutzerdefinierten Dialoge wurden, wie auch die Wizards, mit Hilfe der JFace-Bibliothek erstellt um die gewünschten Funktionalitäten, die (fast) ausschließlich modellbezogen sind, einbringen zu können.

### 3.3.10 Commands und Actions

Commands (engl. Kommandos) und Actions (engl. Aktionen) sind zwei verschiedene APIs mit der selben Aufgabe: Funktionen zu deklarieren und zu implementieren, die der Oberfläche im Sinne von Menüs oder Toolbar-Einträgen zur Verfügung gestellt werden. Wie auch alles andere, werden Commands und Actions mit Hilfe von verschiedenen Extension Points definiert. Die Action API bietet die Möglichkeit verschiedene Extension Points zur Positionierung zu erstellen, beinhaltet aber keine Separierung zwischen der Darstellung und der Implementierung. Die Command API wird über drei verschiedene Extension Points in das System eingebunden, eine zur Definition des Commands, eine zur Positionierung innerhalb der UI und eine zur Spezifizierung der Implementierung. [3, Seite 215ff.]

Zusätzlich zu einem größeren deklarativen Syntax macht dies die Command API flexibler als die ältere Action API[18], und gilt als die bevorzugte Wahl bei der Umsetzung. Es ist ebenfalls möglich den Aktivierungsstatus und die Sichtbarkeit über verschiedene Filter zu kontrollieren. [3, 215ff.]

[18] Die Action API existiert schon seit vor Eclipse 3.0, wohingegen die Command API erst mit Eclipse 3.3 eingeführt wurde. Es ist abzusehen, dass die Action API in der Zukunft als veraltet gekennzeichnet werden wird.

# 4 Implementierungsdetails

## 4.1 Allgemein

Alle Plugins werden unter einem eigenen Namespace „*de.bioforscher.semantec.stf*“[19] geführt. Bei den nachfolgenden Bezeichnungen wird dieser Namespace, zu Gunsten der Übersichtlichkeit, nicht explizit erwähnt. Das zu erstellende System ist skizzenhaft in Abbildung 4.1 dargestellt.

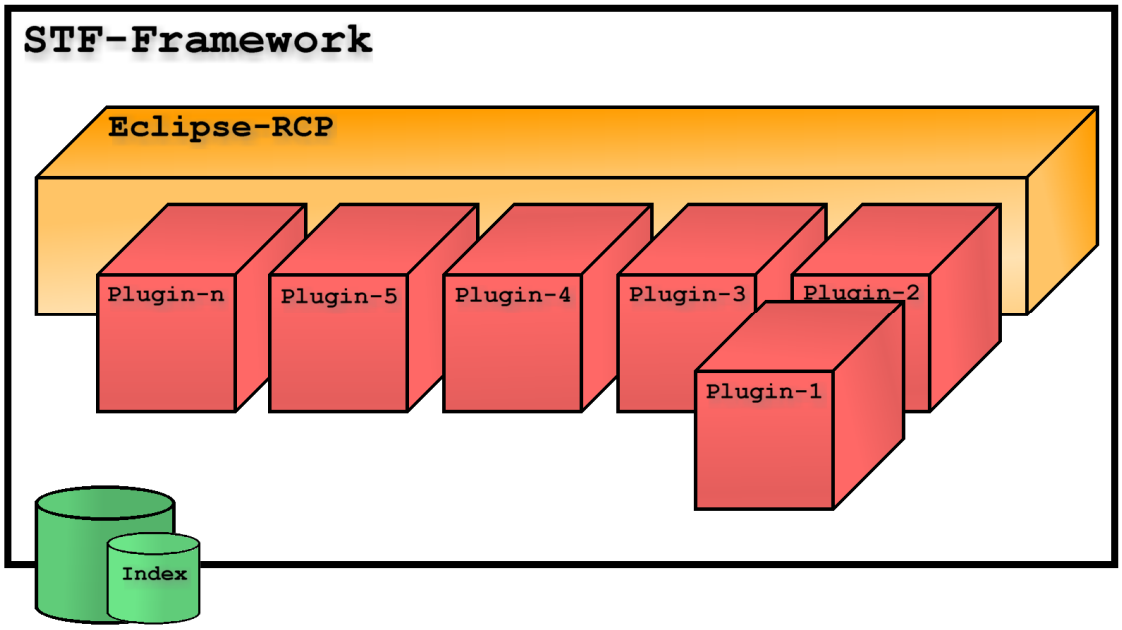


Abbildung 4.1: Systemskizze des zu erstellenden Frameworks

### 4.1.1 Abhängigkeiten

Es wurden verschiedene Plugins erstellt, welche in verschiedene Kategorien eingeordnet werden können. Es erfolgt eine horizontale Teilung innerhalb der einzelnen Modelle, sowie eine vertikale Teilung in Modell und Oberfläche. Zusätzlich sind Standardkomponente vorhanden, welche modellunabhängig sind. Die vertikale Teilung ermöglicht es das vorhandene Modell über verschiedene Oberflächenkomponenten unterschiedlich zu verarbeiten und darzustellen.

[19] „*de.bioforscher*“ stellt die Domain dar, „*semantec*“ steht für „Semantische Technologien“ und spezifiziert das Arbeitsfeld innerhalb der Domain, „*stf*“ bildet den Identifikator für das Projekt

In Abbildung 4.2 sind die Abhängigkeiten der einzelnen Plugins vereinfacht dargestellt. Für eine detailliertere Übersicht der Abhängigkeiten siehe Abbildung A.2.

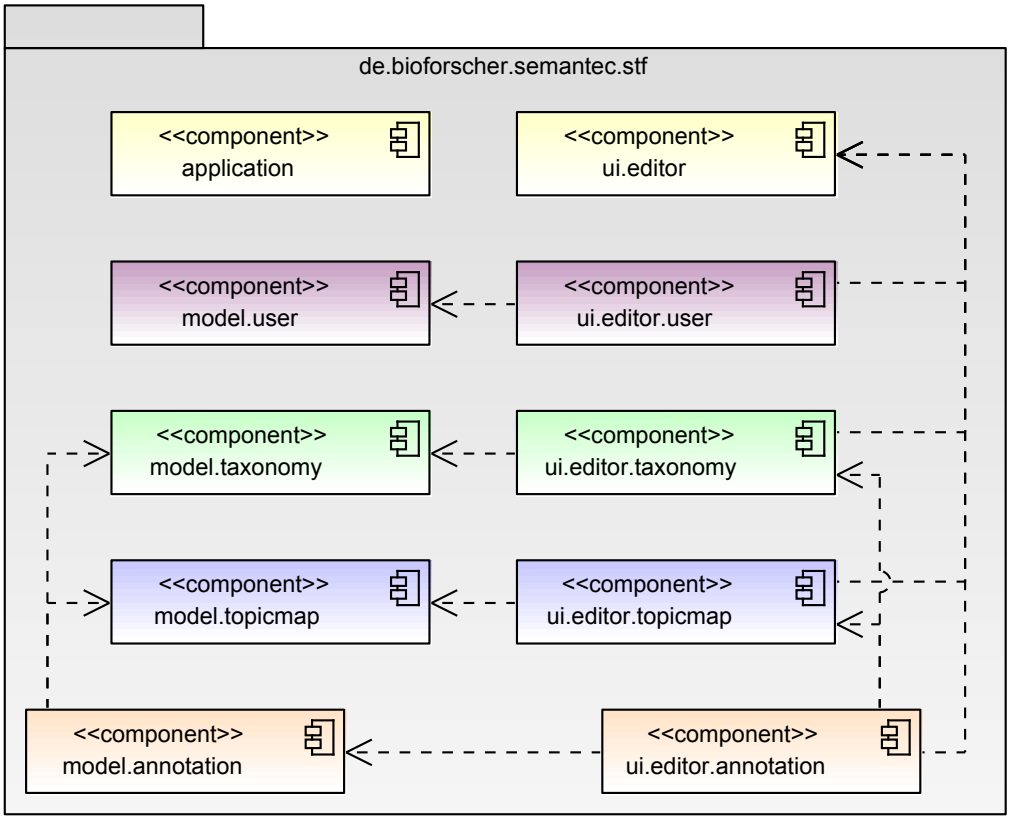


Abbildung 4.2: Übersicht über die Abhängigkeiten der erstellten Plugins

Das Plugin „*application*“ bildet das Grundkonstrukt des Systems. Es werden Einstellungen, Aussehen und Funktionalität von allgemeinen Systemkomponenten definiert. Diese Kernkomponente bildet das eigenständige Rahmenwerk für die Anwendung.

Das Plugin „*ui.editor*“ enthält abstrakte Komponenten zur vereinheitlichten Repräsentation der Editoren. Ein benutzerdefinierter, standardisierter Viewer für die Editoren ist ebenfalls in diesem Plugin enthalten. Alle erstellten Editoren nutzen die Komponenten aus diesem abstrakten Paket.

Der Editor für die Nutzerverwaltung „*ui.editor.user*“ greift auf das erstellte Nutzermodell „*model.user*“ zu und modifiziert es. Eine Erstellung von Taxonomie und Ontologie erfolgen nach dem gleichen Nutzen von Editor und Modell in den Plugins *model.taxonomy* und *ui.editor.taxonomy*, sowie *model.topicmap* und *ui.editor.topicmap*.

Der Annotationseditor „*ui.editor.annotation*“ bedient sich nicht nur dem Annotationsmodell „*mode.annotation*“, sondern auch Teilen der Taxonomie- und Ontologie-Editoren. Da die Annotation sowohl die Taxonomie als auch eine Ontologie im Modell verwendet, können einzelne Komponenten wiederverwendet werden und müssen nicht dupliziert

werden. Eine Modifikation durch Ableitung bestehender Klassen ist problemlos möglich.

### 4.1.2 Das Anwendungs-Plugin

Das Plugin „application“ beinhaltet alle zum Anwendungsstart notwendigen Dateien, Klassen und Einstellungen. Dazu gehört eine Klasse mit der *IApplication*-Schnittstelle, welche als Extension (*org.eclipse.core.runtime.applications*) eingebunden und referenziert wird. Diese *Application*-Klasse kontrolliert alle Aspekte der Anwendungsausführung. Sie unterliegt nicht dem Prinzip des „Lazy Loading“ und kann diesbezüglich Menü- und Toolbar-Einträge programmatisch anpassen und auf die Benutzung von Extension Points verzichten. Weiterhin ist ein *WorkbenchAdvisor* vorhanden, welcher durch die *Application* gestartet wird und selbst den *WorkbenchWindowAdvisor* initialisiert. Die beiden „Berater“ konfigurieren das Aussehen der Anwendung bzw. des Fensters im Sinne von Position, Größe, Menüs, Toolbars, sowie der initialen Perspektive. Die Perspektive legt das Layout der Oberflächenkomponente fest – so erhält der Bereich für die Editoren seinen Platz, bzw. werden die Views und die Plazthalter für Views um den Editorbereich angeordnet.

Es wird vorerst nur ein für das Projekt relevanter View für die Navigation zwischen den Projekten fest gesetzt. Der Navigator ist mit Hilfe des „*Common Navigator Framework*“ (CNF) erstellt und bietet dadurch eine sehr hohe Flexibilität für verschiedene Daten. Es ist nicht nur möglich einen Content Provider, Label Provider, Sorter und Filter für den View zu benutzen, das CNF erweitert diese Möglichkeit um dynamisch mehrere Provider, Sorter und Filter bereit zu stellen. Diese können über den Extension Point „org.eclipse.ui.navigator.viewer“ bestimmt und auf Daten angewendet werden, die bestimmten Mustern entsprechen. [7]

Für die Anwendung werden zusätzlich noch ein Startbildschirm (Splash-Screen) und eine Willkommensseite (Intro) erzeugt, welche mit Hilfe von Extensions eingepflegt und konfiguriert werden. Der Startbildschirm kann frei konfiguriert oder von einem bestehendem Template generiert werden. Für das Projekt wurde ein Login-Template verwendet, welches die Nutzeranmeldung durchführt und anschließend entsprechende Plugins lädt. Dafür wurde ein interaktiver Splash-Handler erzeugt, der die Logik der Anmeldung übernimmt und von *AbstractSplashHandler* abgeleitet ist. Als einfache Willkommensseite wurde die Webseite „bioforscher.de“ gewählt um aktuelle Informationen zum Projektstand darzustellen. Diese beiden Brandings werden mit Hilfe eines Produktes für die Anwendung bereit gestellt.

Das Produkt selber ist ebenfalls nur eine Extension zum Extension Point „*org.eclipse.core.runtime.products*“ mit zusätzlichen Eigenschaften. Diese definieren das Anwendungsspezifische Branding über allen Plugin-Konfigurationen. Es spezifiziert den Iden-

tifikator (ID) der Anwendung mit dem es assoziiert ist und stellt einen Namen, eine Beschreibung und eine Versionsnummer bereit. Zusätzlich können weitere Eigenschaften abgelegt werden, wie zum Beispiel das Icon der Anwendung. [8]

Ein spezieller Fall, bei dem die Nutzung von Extension Points essentiell ist, besteht in der aktuellen Version von Eclipse RCP, bei den Activities. Eine Activity ist eine logische Gruppierung von Funktionen, die um einen bestimmten Aufgabenkomplex angeordnet sind. Beispielsweise werden bei der Etablierung der Java-Entwicklungsumgebung für Eclipse viele Oberflächenelemente von der JDT hinzugefügt, welche nur sinnvoll verwendet werden können, wenn eine Java basierte Programmentwicklung stattfindet. Eine weiterer Nutzen von Activities[20] besteht darin, bestehende Oberflächenelemente nach bestimmten Kriterien, wie den Zugriffsrechten eines Nutzers, auszufiltern. [6] Der daraus resultierende Vorteil ist ersichtlich – es können beispielsweise rollenbasiert Oberflächenveränderungen getroffen werden. Es ist allerdings weiterhin möglich Actions und Commands über registrierte Tastaturkürzel aus den entsprechend ausgeblendeten Oberflächenelementen zu nutzen. Eine Separierung von Nutzerprofilen auf dem Weg von Activities ist also nicht empfehlenswert. Das Projekt verwendet dennoch Activities, allerdings nur zum Verbergen von standardisierten Oberflächenelementen, die durch importierte Abhängigkeiten auftauchen. Die Implementierung erfolgt über den Extension Point „org.eclipse.ui.activities“. [29]

Eine andere Möglichkeit Aktionen für die Oberfläche in Übersichtlicher Form zugänglich zu machen, besteht in der Verwendung von *ActionSets*. Ein *ActionSet* gruppiert eine Menge von Aktionen unter einem Identifikator zusammen, um problemspezifische Aufgaben gemeinsam verwalten zu können. Zusätzlich zur Verwendung von Activities zum Verbergen von Oberflächenelementen, werden *ActionSets* aus der Oberfläche entfernt, die durch importierte Abhängigkeiten für das Projekt unnütze Funktionen in Menüstrukturen eintragen. Das Entfernen der *ActionSets* wird mit Hilfe des *WorkbenchWindowAdvisor* auf programmatischem Wege durchgeführt.

### 4.1.3 Das Editor-Plugin

Das Plugin „ui.editor“ (Editor-Plugin) beinhaltet eine Reihe von abstrakten Klassen und Templates zur Vereinheitlichung der Oberflächenelemente und zur Vereinfachung und Reduzierung der Redundanz des Quellcodes. Als Grundlage für die Oberfläche des Editor-Plugins wird die Bibliothek „Eclipse Forms“ eingebunden und verwendet.

Eclipse liefert mit den *Eclipse Forms* ein Web-artiges Aussehen der Benutzeroberflächen. Eclipse Forms („org.eclipse.ui.forms“) ist ein auf SWT und JFace basierendes Plugin zur Erzeugung von portierbaren, web-artigen Benutzeroberflächen über alle UI Kategorien hinweg. Dabei passt sie das Aussehen und Verhalten („Look and Feel“) der Stan-

[20] Diese Möglichkeit existiert seit Eclipse 3.4

dardelemente an die Forms API an und erzielt das Aussehen der Oberfläche ähnlich einer Webseite. Die Klasse FormToolkit dient als Factory zur Erzeugung von benötigten UI-Elementen. Bereits existierende Elemente können über die `adapt(Composite)`-Methode ebenfalls an die Forms API angepasst werden. [32]

Es ist möglich das Editor-Plugin in vier funktionelle Bereiche zu Gliedern, den Editor, den Wizard, den Viewer und die Commands.

#### 4.1.3.1 Der abstrakte Editor

Der abstrakter Editor dient als Grundlage für das Aussehen und die Basisfunktionalitäten von darauf aufbauenden Editoren zur Gestaltung einer benutzerfreundlichen, einheitlichen Oberfläche. Es wurde darauf acht gegeben, eine möglichst offene Schnittstelle zu generieren, um Platz für Erweiterungen nicht zu versperren und gleichzeitig eine notwendige Abgeschlossenheit zu bieten, um Redundanz möglichst gering zu halten.

Das Editor-Template besteht aus fünf hierarchisch angeordneten Klassen, wie in der Abbildung 4.3 dargestellt. Die ausgezeichneten Operationen innerhalb der Klassen bilden die abstrakten Operationen, die von abgeleiteten Klassen überschrieben werden müssen.

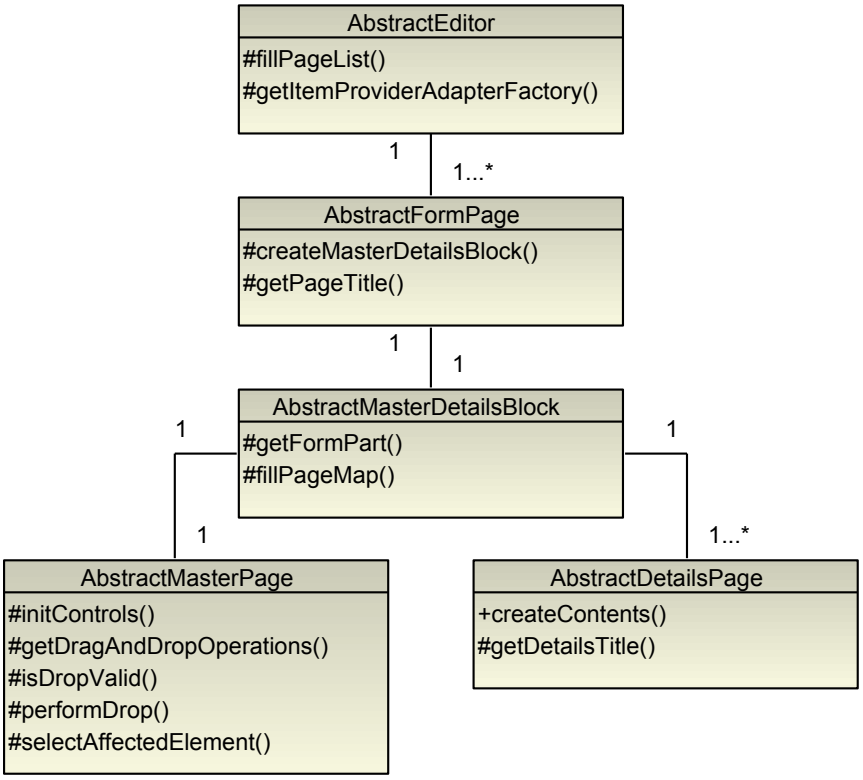


Abbildung 4.3: Klassenhierarchie des abstrakten Editors

**AbstractEditor**
Als Einstiegspunkt wird die Klasse *AbstractEditor* mit dem Interface *IEditorPart* definiert. Durch die Verwendung der Forms API kann der abstrakte Editor direkt von *FormEditor* abgeleitet werden, welcher das Interface ebenfalls implementiert. Diese Klasse, insbesondere das Interface, wird später im Extension Point „org.eclipse.ui.editors“ zur Registrierung benötigt.

Der Editor übernimmt die Aufgabe der zentralen Verwaltung des Modells in Form der Erzeugung, Initialisierung, Validierung und Persistierung der EditingDomain bzw. der entsprechenden Ressourcen. Durch die Verwendung des Interface *IEditingDomainProvider* stellt es gleichzeitig die EditingDomain als Singleton für alle Interessenten bereit. Um eine flexible Bearbeitung des Modells der EditingDomain zu gewährleisten, muss über die Methode `getItemProviderAdapterFactory()` ein entsprechender Provider für das Modell von einer Unterklasse implementiert werden.

Die Validierung des Modells erfolgt über einen *EContentAdapter* an der EditingDomain, welcher Modell- und Ressourcenprobleme auswerten, klassifizieren und darstellen kann.

Jede Unterklasse muss die Methode `fillPageList()` implementieren, um einzelne Seiten (*FormPage*) innerhalb des Editors zu registrieren. Diese werden während der Initialisierung des Editors dann entsprechend eingehängt und zur Verfügung gestellt.

Der abstrakte Editor übernimmt zusätzlich die Aufgabe einen *SelectionProvider* zu registrieren. Dieser wird als zentrales Verwaltungsmedium für ausgewählte Objekte verwendet. Es können Benachrichtigungen über Änderungen in der Auswahl eines Objektes, über die Registrierung eines *Listeners*, an andere Komponente (z.B.: Views) gegeben werden.

**AbstractFormPage**
Ein *FormEditor* kann mehrere Klassen mit der Schnittstelle *IFormPage* als Seiten bzw. Reiter aufnehmen. Die *AbstractFormPage* implementiert dieses Interface über die Oberklasse *FormPage*.

Im Konstruktor von *AbstractFormPage* wird der *AbstractEditor* als Referenz übergeben und ist über die Methode `getEditor()` abrufbar.

Jede Unterklasse der abstrakten Seite muss einen Titel zur Identifizierung (`getPageTitle()`), sowie einen angepassten MDB (`createMasterDetailsBlock()`) implementieren. Sollte es vorkommen, dass die Erzeugung eines MasterDetailsBlock nicht gewollt ist, um andere Elemente auf der Seite unterzubringen, so kann die Methode `createFormContent()` überschrieben werden.

**AbstractMasterDetailsBlock**

Standardmäßig ist vorgesehen, dass eine *AbstractFormPage* als füllendes Hauptelement einen Master-Details-Block in Form der Klasse *AbstractMasterDetailsBlock* erhält.

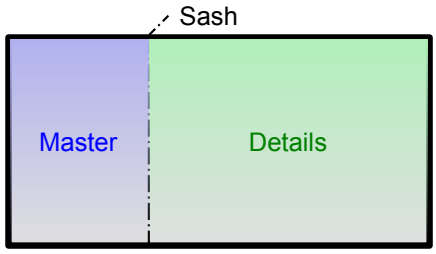


Abbildung 4.4: Schematischer Aufbau eines Master-Details-Block

Das Oberflächen-Muster Master-Details-Block (MDB) besteht aus zwei Teilen, dem „Master“ und den „Details“ innerhalb einer SashForm (engl. Schiebefenster), welches zur Veränderung des Verhältnisses der beiden Teile zueinander innerhalb der Seite verwendet werden kann (siehe Abbildung 4.4). Der „Master“ wird als bestehende Komponente erzeugt, während die „Details“ am MDB statisch oder dynamisch registriert werden und später durch diesen automatisch nach Bedarf erzeugt werden. Der „Master“ ist dafür verantwortlich Selection-Events auszulösen, welche vom „Details“-Teil ausgewertet werden können, um eine Seite zu laden, die mit dem entsprechenden Objekt assoziiert wurde. Die Seite kann verwendet werden, um Details zum ausgewählten Objekt anzuzeigen und diese zu bearbeiten.

Um einen entsprechenden Master-Teil zu generieren muss die `getFormPart()`-Methode überschrieben werden und für die Registrierung von Detail-Seiten auf Objekte muss `fillPageMap()` implementiert werden.

**AbstractMasterPage**

Der Master-Teil wird, entsprechend des vorhandenen Modells, mit Daten versorgt, die beliebig dargestellt werden können. Üblicherweise handelt es sich dabei im Projekt um komplexere Modelle, die in Form von Listen, Tabellen oder Bäumen dargestellt werden. Der abstrakte Master-Teil *AbstractMasterPage* für den MDB wird deswegen standardmäßig mit einem „*ControlledViewer*“ versehen. Es handelt sich dabei um einen modifizierten Viewer, auf dem im nächsten Abschnitt 4.1.3.2 näher eingegangen wird. Damit dieser generell benutzt werden kann, müssen Unterklassen `initControls()` überschreiben. Sollen andere Komponenten im Master-Teil auftauchen, können sie ebenfalls in dieser Methode hinzugefügt werden, zusätzlich müssen dann aber die Methoden `refreshViewer()`, `setFormInput()` und `refresh()` überschrieben werden.

Der Viewer unterstützt die grundsätzliche Verwendung von „Drag and Drop“ (DND). Über `getDragAndDropOperations()` kann die Verwendung durch Unterklassen konfiguriert werden. Liefert diese Methode etwas anderes als die Konstante „*DND.DROP_*

*NONE*“[21], so wird während des Drag & Drop-Vorgangs mittels `isDropValid()` das zu betrachtende Objekt, sowie der Zielort evaluiert und beim Abschließen des Vorgangs die Methode `performDrop()` aufgerufen. Die beiden zuletzt genannten Methoden müssen von Unterklassen implementiert werden, auch wenn kein Drag & Drop konfiguriert wurde.

Werden Änderungen am Modell durchgeführt, wird der Master-Teil über die Änderungen mit Hilfe eines CommandStack-Listeners darüber informiert. Wie auf diese Modifikation reagiert werden soll, kann mittels `selectAffectedElement()` implementiert werden. Auf diese Weise ist es beispielsweise möglich ein neu generiertes Objekt im Modell zu selektieren und daraus resultierend gleichzeitig die entsprechende Details-Seite zu laden bzw. zu aktualisieren.

**AbstractDetailsPage**
Eine abstrakte Details-Seite *AbstractDetailsPage* für den MDB wird auf Grund der Verschiedenartigkeit der darzustellenden Informationen der Modellobjekte sehr allgemein gehalten.

Abgeleitete Klassen müssen die Methode `getDetailsTitle()` überschreiben, um der Detail-Seite einen Namen bzw. eine Überschrift zu geben und können durch die Implementierung von `createContents()` beliebige Oberflächenkomponenten erzeugen. Sobald ein SelectionChanged-Event, ausgelöst durch den Master-Teil des MDB, auftritt, kann auf dieses mittels `selectionChanged()` reagiert und die Oberfläche aktualisiert werden. Um eine Strukturierung innerhalb von Detail-Seiten zu ermöglichen, können „*Sections*“ mit Hilfe der Methode `addSection()` generiert werden.

Eine *Section* ist ein ausklappbares Container-Element für weitere Komponenten, welches optional eine Überschrift, eine Beschreibung und eine Toolbar besitzen kann. Es wird häufig als ein Mittel zur logischen Gruppierung von Informationen verwendet und bildet eine Grundkomponente innerhalb der Forms API.

#### 4.1.3.2 Der filterbare Viewer

Die Verwendung von JFace-Viewern ist für eine komfortable Darstellung und Modifikation von komplexeren Daten optimal. Für eine schnelle Suche nach einzelnen Datensätzen in einem großen Datenbestand bieten Viewer durch die Verwendung eines *ViewerFilter* die Möglichkeit die Datensätze anhand von Mustern zu filtern. Um die Redundanz bei der Implementierung einer Oberflächenkomponente, die diesen Filter umsetzt, zu minimieren, wurde ein benutzerdefiniertes UI-Element *ControlledViewer* erstellt, welches projektweit verwendet wird.

[21] Mögliche Konfigurationen sind die Werte „*DND.DROP_COPY*“, „*DND.DROP_LINK*“ und „*DND.DROP_MOVE*“ bzw. bitweise Oder-Verknüpfungen dieser Konstanten.

Zur Umsetzung des Filters ist die Eingabe eines Textmusters[22] erforderlich. Es wurde dabei das SWT Text-Widget mit der Möglichkeit der Eingabe von modifizierbarem Text gewählt. Die abstrakte Filter-Klasse *AbstractControlledFilter* erweitert den *ViewerFilter* und implementiert einen *ModifyListener*. Das Text-Widget registriert den Listener der Filter-Klasse zur Überwachung auf Inhaltsänderungen und der Viewer registriert die Filter-Klasse als *ViewerFilter*. Ändert sich der Inhalt des Text-Widgets, informiert der Listener registrierte Viewer über eine Änderung, nachdem in einem Pre-Prozess das Filtermuster in einen regulären Ausdruck umgewandelt wird, falls es noch nicht in dieser Form vorliegt. Die Viewer aktualisieren sich komplett und beziehen die Modelldaten neu, wobei der Filter-Prozess angestoßen wird. Den Vergleich muss jede abgeleitete Klasse mit Hilfe der `select()`-Methode je nach Modell selbst implementieren. Mit Hilfe von `matchesFilter()` kann innerhalb von `select()` ein Textstring mit dem Filter abgeglichen werden. Besteht ein einzelnes Datenobjekt den Filter-Test nicht, so wird es in der Oberfläche nicht angezeigt.

Handelt es sich bei dem Viewer-Typ um eine baumartige Darstellung, so werden neben dem Text-Widget noch zwei weitere Button-Widgets in einer Toolbar erzeugt. Mit Hilfe dieser Schaltflächen können alle Knoten im Baum ein- bzw. ausgeklappt werden. Dabei werden die nativen Funktionen des Viewers verwendet.

Die Implementierung der Widgets innerhalb des Viewers oder innerhalb der gekapselten SWT-Widgets ist auf Grund von Restriktionen bei der Ableitung dieser Klassen nicht möglich. Es wurde eine Container-Komponente (Composite) verwendet, auf die die Widgets und ein entsprechender Viewer gesetzt wurden. Bei der Initialisierung dieser Komponente muss die Art des Viewers und ein *AbstractControlledFilter* angegeben werden.

#### 4.1.3.3 Die Standard-Commands

Unabhängig von der Art des abgeleiteten Editors, gibt es zwei Aktionen, die grundsätzlich in jedes Kontextmenü innerhalb des Editors eingebunden werden sollen. Dabei handelt es sich um eine Redo-, sowie eine Undo-Funktion – das Wiederherstellen und Rückgängig machen der letzten Aktionen.

Die Verwaltung der EditingDomain innerhalb des Editors und die damit verbundene Bereitstellung des CommandStacks ermöglichen eine zentrale Redo-Undo-Funktionalität auf der Basis der verwendeten EMF Commands. Die Implementierung dieser Funktionen und die Bereitstellung der Menüeinträge werden über die UI Commands realisiert. Dazu ist es notwendig verschiedene Extension Points zu verwenden.

Die UI Commands müssen über den Extension Point „org.eclipse.ui.commands“ defi-

[22] Die Filterfunktion unterstützt die Verwendung von regulären Ausdrücken (RegEx) zum Abgleichen der Quell- und Zieldaten.

niert werden, in dem sie einen Namen sowie eine eindeutige ID erhalten. Nun ist es möglich diese generierten Aktionen über den Extension Point „org.eclipse.ui.menu“ in ein Menü einzutragen. Dazu werden vorerst alle Popup-Menüs der Oberfläche über die Konstante „popup:org.eclipse.ui.popup.any“ angesprochen, mit dem Zusatz, dass die erzeugten Elemente nach dem Element „*additions*“ innerhalb eines bestehenden Menüs eingetragen werden sollen („?after=additions“). Die beiden UI Commands werden innerhalb von sichtbaren Separatoren hervorgehoben und mit einem Standard-Icon versehen. Um die UI Commands mit einer Funktion zu assoziieren, ist es notwendig deren Logik mit Hilfe von Handlern über den Extension Point „org.eclipse.ui.handlers“ zu implementieren.

Ein Handler wird für ein bestehendes UI Command definiert und ihm wird die Referenz einer Klasse übergeben, die das Interface „IHandler“ implementiert. Dem deklarativem Handler können weitere Eigenschaften bezüglich der Sichtbarkeit (visibleWhen) und der Aktivierung (enableWhen) zugeordnet werden. Diese Eigenschaften können mit Hilfe von definierten Ausdrücken spezifiziert werden. Im Fall der Redo-Undo-Funktionen wird die Aktivierung einer entsprechenden Methode geprüft, falls ein Menü auf einem bekannten Modellelement geöffnet wurde. Ob es sich um ein bekanntes oder unbekanntes Element handelt kann mit Hilfe eines *PropertyTester* evaluiert werden. Der *PropertyTester* wird mit Hilfe des Extension Points „org.eclipse.core.expressions.propertyTester“ definiert. Dazu erhält er eine eindeutige ID, einen Objekttyp, Attribut-Namen des Objekts, die geprüft werden sollen, einen Namensraum, mit dem die Attribute innerhalb der Extension Points identifiziert werden können, sowie eine Klasse, die das Interface „IPropertyTester“ implementiert. Der Namensraum und die entsprechenden Attribute werden innerhalb der Ausdrücke für die Aktivierung der Menüelemente referenziert. Sobald ein Objekt mit dem entsprechenden Attribut mit dem Menü assoziiert wird, so wird die Logik (die Klasse) des *PropertyTester* ausgeführt.

Innerhalb der Klasse des *PropertyTester* kann mit der Methode `test()` das aktuelle Objekt validiert werden. Im Anwendungsfall für die Redo-Undo-Funktionen ist es sogar irrelevant welches Objekt getestet wird. Es wird der aktuelle Editor des Objektes nach der EditingDomain abgefragt und der CommandStack bezogen. Der Rückgabewert der `canRedo()`- bzw. `canUndo()`-Funktion des CommandStacks wird als Validator für einen erfolgreichen Test verwendet und aktiviert oder deaktiviert dementsprechend den jeweiligen Menüeintrag.

Die mit dem Handler referenzierte Logik-Klasse wird beim Ausführend des Menüeintrags aktiviert. Für eine Abwärtskompatibilität und höhere Flexibilität, sowie einer Reduzierung der Redundanz wurde eine abstrakte Oberklasse *AbstractCommandHandler* erzeugt, die das Interface „IHandler“ implementiert. Der abstrakte Handler übernimmt die Extraktion der EditingDomain aus dem aktuellen Editor und delegiert bei Erfolg diese und weitere Attribute an die abstrakte Methode `handleCommand()`. Die Command UI kann von diesem Handler abgeleitete Klassen problemlos nutzen und imperative Aufrufe

können ebenfalls über die `handleCommand()`-Methode erfolgen. Auf diese Weise kann ein deklaratives UI Command zusätzlich aus dem Quellcode heraus auch an anderen Stellen verwendet werden. Für die Redo-Undo-Funktionalität wird die entsprechende Funktion des CommandStacks ausgeführt, der über die übergebene EditingDomain bezogen werden kann.

#### 4.1.3.4 Der abstrakte Wizard

Bevor die Modelle mit Editoren bearbeitet oder über andere Schnittstellen modifiziert werden können, müssen sie erzeugt und initialisiert werden. Dieser Schritt erfolgt mit Hilfe von Wizards, bzw. einem abgeleiteten Wizard von *AbstractWizard*, welcher die initiale Modellgenerierung übernimmt. Mit Hilfe des abstrakten Wizards soll eine einheitliche Erstellung der semantischen Konzepte ermöglicht werden. Dazu ist es erforderlich ein Zielprojekt, sowie einen Ressourcenname zu wählen, das Modell zu initialisieren und evtl. ein auswählbares Template als Grundlage zu verwenden.

Im ersten Schritt soll ein bestehendes Projekt aus dem Arbeitsbereich gewählt und ein Name für die neue Ressource vergeben werden. Die Startseite des Wizards ist vom Typ eine *AbstractCreationWizardPage*, welche diese Aufgabe übernimmt. Ist es erforderlich, so können Methoden dieser Seite überschrieben werden, ansonsten übernimmt sie alle Aufgaben ohne weitere Konfiguration. Als einzige Restriktion gilt die Namenskonvention, dass keine zwei identischen Ressourcen-Namen innerhalb eines Projekts vergeben werden dürfen. Eine Instanz der ersten Seite muss über `getCreationPage()` implementiert werden.

Im zweiten Schritt ist es optional möglich, ein Template des semantischen Konzepts zu wählen. Es können nur Templates des gleichen Konzepts gewählt werden, was anhand der Dateiendung geprüft wird. Die Verwendung der abstrakten Seite *AbstractTemplateWizardPage* übernimmt die Grundfunktionalitäten und den Aufbau der Template-Auswahl und wird im Wizard durch die `getTemplatePage()`-Methode bereit gestellt. Um einen Spielraum für zusätzliche Optionen bereit zu stellen, kann die Methode `createOptionsArea()` befüllt und mit `applyOptions()` validiert werden. Auf Grund verschiedener Modelle, die über den abstrakten Wizard angesteuert werden können, müssen zusätzlich Adapter (`getItemAdapterFactory()`), Provider (`getContentProvider()`, `getLabelProvider()`) und Filter (`getFilter()`) implementiert werden. Eine zusätzliche Modifikation ermöglicht das abwählen von einzelnen Modellkomponenten, die aus dem Template nicht mit übernommen werden sollen. Elemente, die auf jeden Fall aus einem Template übernommen werden müssen, können über `canDelete()` konfiguriert werden.

Sobald der Wizard fertig gestellt wird, erzeugt er alle notwendigen Ressourcen. Ist ein Template vorhanden, so werden die Optionen der *AbstractTemplateWizardPage* angewendet, falls welche vorhanden sind. Anschließend werden die markierten Elemente

des Modells in die neue Ressource kopiert. Ist kein Template ausgewählt, so muss das Modell neu erzeugt werden. Je nach Art des Modells muss die abgeleitete Klasse `createInitialModel()` im *AbstractWizard* implementieren.

## 4.2 Modellspezifische Implementierungen

Die Entwicklung von modellspezifischen Editoren orientiert sich stark an der Ableitung und Umsetzung der abstrakten Klassen des Editor-Plugins. Es wurde für jedes Modell ein Editor erstellt und in zwei Plugins verteilt, das Kern-Plugin „ui.editor.*modellname*“ und ein erweiterndes Plugin „ui.editor.*modellname*.util“ (Utility-Plugin).

Das Kern-Plugin jedes Modells umfasst die deklarativen Beschreibungen und damit die Extensions, die dieses Plugin bereit stellt, sowie die eigentlichen Editor-Klassen. Das Utility-Plugin beinhaltet verschiedene Klassen, die eine losgelöste Logik des Plugins repräsentieren, darunter fallen Wizards, Dialoge, Provider, CommandHandler und Filter. Diese werden vom Kern-Plugin benötigt um die entsprechenden Funktionalitäten anbieten zu können, sind selbst aber nur auf das jeweilige Modell angewiesen und können daraus resultierend auch ihre Funktionalität anderen Plugins bereit stellen, die auf dem Modell arbeiten.

Jedes Modell verfügt zusätzlich über eine Hilfsklasse, die Funktionalitäten nach außen hin kapselt und vereinfacht. Diese Klasse wird nicht durch EMF generiert und befindet sich im Modell-Plugin „de.bioforscher.semantec.stf.model.*modellname*“ im Paket „util.factory“.

Implementierungsdetails der einzelnen Plugins werden in den nachfolgenden Abschnitten näher erläutert.

### 4.2.1 Modell-Werkzeuge

Die Modell-Hilfsklassen bieten statische Fabrikmethoden an, um Modellelemente mit geringem Aufwand zu erzeugen und zu initialisieren. Dabei wird die reflektive API von EMF verwendet, auch wenn sie an dieser Stelle durch die feste Verdrahtung im Code, ihr eigentliches Potential nicht ausschöpfen kann. Die Klasse *TaxonomyUtil* demonstriert dies am Beispiel der Instanziierung eines *CategoryDescriptor*-Elements einer Taxonomie in der Abbildung 4.5.

Die Hilfsklassen stellen zusätzlich Funktionalitäten bereit um komplexere Strukturen und Abhängigkeiten von Modellelementen zu beziehen und aufzubereiten. So können beispielsweise alle Kind-Elemente[23] einer Kategorie der Taxonomie, rekursiv bis zu den Blättern des Baumes, bezogen werden.

Im dritten großen Funktionskomplex werden Teilevaluierungen und -validierungen der Modelle durchgeführt. Es können Abfragen an das Modell gestellt werden, um eine Vollständigkeit oder Korrektheit der angegebenen Daten zu prüfen. In diesem Zusam-

[23] Die *EObject*-Funktionen bieten nur an direkte Kindelemente zu beziehen.

```
public static CategoryDescriptor createNewCategoryDescriptor()
{
    TaxonomyPackage p = TaxonomyPackage.eINSTANCE;
    TaxonomyFactory f = p.getTaxonomyFactory();

    return (CategoryDescriptor) f.create(p.getCategoryDescriptor());
}
public static CategoryDescriptor createNewCategoryDescriptor(String name)
{
    CategoryDescriptor cat = createNewCategoryDescriptor();
    cat.eSet(TaxonomyPackage.Literals.CATEGORY_DESCRIPTOR__NAME, name);

    return cat;
}
```

Abbildung 4.5: Quellcodeausschnitt: Erzeugung und Initialisierung eines Modellelements

menhang ist es möglich Fehlerquellen während der Bearbeitung des Modells hervorzuheben und Benutzerinteraktionen anzufordern, bevor das Modell bei der Persistierung durch den Editor geprüft wird.

### 4.2.2 Taxonomie

Der Taxonomie-Editor leitet den abstrakten Editor mit seinem Standardverhalten ab und implementiert notwendige Methoden. Das Taxonomie-Modell wird seiner Natur nach in einer Baumstruktur in der Master-Seite des MDB visualisiert. Es wird eine Detail-Seite erzeugt und mit dem einzigen in diesem Editor bearbeitbaren Objekt *Category-Descriptor* assoziiert. Innerhalb der Detail-Seite werden nur Objektinformationen präsentiert. Die Erzeugung und Modifikation von Kategorien der Taxonomie erfolgt über Kontextmenü-Interaktionen im Viewer des Master-Teils.

Die Aktionsmöglichkeiten reichen vom Erzeugen einer neuen Taxonomie mit einem entsprechenden Wizard, abgeleitet vom abstrakten Wizard, über das Hinzufügen und Verschieben von Kindelementen bis zum Löschen von einzelnen Elementen. Die einzige Besonderheit liegt darin, dass das feste Wurzelelement nicht gelöscht werden darf, was über einen PropertyTester sichergestellt wird.

Das Utility-Plugin der Taxonomie bietet zusätzlich noch einen Provider für die Visualisierung von *DocumentDescriptor*-Objekten innerhalb der Kategorien an, welcher aber vom Kern-Plugin nicht in Anspruch genommen wird.

### 4.2.3 Topic Map

Das wesentlich komplexere Modell der Topic Map gegenüber der Taxonomie spiegelt sich auch in der Implementierung der Topic Map-Plugins wieder. Die Visualisierung der

Modellbestandteile wurde in zwei Ebenen verlagert.

**Master- und Detail-Seiten**

In der ersten Ebene werden die Topics vom Typ „Topic“, „Association“ und „Association-Role“ als Kernkomponenten dargestellt und verwaltet. Um die logische Auftrennung und eine bessere Übersichtlichkeit zu gewähren, wird für jeden dieser Topic-Typen ein eigener Viewer innerhalb der Master-Seite des MDB bereit gestellt. Organisiert werden die Viewer innerhalb eines „Shelf-Widgets[24]“, einer gestapelten Liste von Elementen, wobei ein Element ausgeklappt werden kann um den damit assoziierten Inhalt zu präsentieren (alle anderen Elemente werden dafür eingeklappt). Zur Verwendung des *Shelf* müssen Anpassungen bei der Master-Seite *TopicmapMasterPage* auf drei Viewer erfolgen. Die Methode `initControls()` erzeugt neben dem Standard-Viewer noch zwei weitere, die durch das Überschreiben von `setFormInput()` ebenfalls mit Modelldaten gespeist und über `refreshViewer()` aktualisiert werden. Jeder der Viewer registriert ein eigenes Kontextmenü, in welches über die Extension Points verschiedene Aktionen eingetragen werden.

Es wird eine Detail-Seite mit dem *Topic*-Objekt assoziiert, welche Objekteigenschaften, wie Identität, Name, Gültigkeitsbereiche, Facetten, Occurrences und Assoziationen übersichtlich präsentiert. Für die Erzeugung und Bearbeitung eines Topics wird ein mehrseitiger Wizard verwendet, der über das Kontextmenü der Viewer der Master-Seite, sowie über Toolbar-Aktionen der Detail-Seite aufgerufen werden kann. Je nach Art des Aufrufs werden über verschiedene Parameter relevante Seiten innerhalb des Wizards angezeigt.

**Wizards und Dialoge**

Die zweite Ebene der Modellvisualisierung besteht aus einer Verbindung zwischen diesem mehrseitigen Wizard *TopicmapWizardTopic* und verschiedenen Dialogen. Die primäre Seite *TopicmapWizardTopicPageName* des Wizards verwaltet die Topicnamen, Namenseinträge und Gültigkeitsbereiche für diese Attribute. Die Angabe einer Identität kann ebenfalls optional erfolgen. Auf der zweiten Seite *TopicmapWizardTopicPageProperties* können Facetten, Occurrences und die Gültigkeitsbereiche für das Topic an sich verändert werden. Diese Angaben sind gegenüber der korrekten Vergabe eines oder mehrerer Namen der ersten Seite alle optional. Die dritte und letzte Seite *TopicmapWizardTopicPageAssociation* dient der Verwaltung der Assoziationen des Topics, durch die Bestimmung einer Assoziation, eines Ziel-Topics und entsprechenden Rollen für die Teilnehmer der Assoziation. Alle Modifikationen (im Sinne von EMF Commands) innerhalb des Wizards werden über ein *CompoundCommand* aufgenommen. Wird der Wizard erfolgreich fertig gestellt, so werden die Änderungen als Ganzes auf den CommandStack gelegt, damit ein Rückgängig machen der letzten Aktion die kompletten Änderungen des Wizards erfassen kann, anstatt nur einzelne Teilschritte zu negie-

[24] Das Shelf-Widget wird über die Nebula-Projektseite bezogen, welche solide, benutzerdefinierte JFace und SWT Widgets durch unabhängige Entwickler bereit stellt. [20]

ren. Kann der Wizard nicht erfolgreich fertig gestellt werden oder wird er abgebrochen, so werden die aufgenommenen Änderungen rückgängig gemacht und verworfen. Die gleiche Vorgehensweise wird auch bei den einzelnen Dialogen angewendet, die für das Projekt implementiert wurden.

Die komplexe modulare Struktur des Topic Map-Modells ermöglicht die Wiederverwendung von Dialogen für verschiedene Aufgabenbereiche. Das Modell-Paradigma, dass jedes Element durch ein Topic repräsentiert wird, kommt dem dialogbetriebenen Vorgehen entgegen. An diversen Stellen der Erzeugung von Modellelementen kommt beispielsweise der Einsatz von Scopes zum tragen, welcher über einen einzigen Dialog realisiert werden kann. Trotz der starken Ähnlichkeit der Einsatzmuster der Dialoge, sind verschiedene Prüfungen auf Korrektheit der Eingaben notwendig. Mit Hilfe einer abstrakten Validierungs-Klasse *AbstractValidator* können spezifische Implementierungen mittels `doValidation()` zur Evaluierung eines Dialogs übergeben werden, die dann bei einer Änderung über `validate()` innerhalb des Dialogs aufgerufen werden können.

Sowohl der Wizard als auch die Dialoge verwenden verschiedene Typen des *Controlled-Viewer* zur Darstellung der einzelnen Modell-Komponenten. Dabei werden eine Vielzahl an Providern für die Inhalte und Label erstellt, die für eine weitere Verwendung über das Utility-Plugin zur Verfügung stehen.

**TopicmapUtil**
Das Modell-Werkzeug *TopicmapUtil* ist vor allem bei der Auswertung der Namenseinträge in den verschiedenen Gültigkeitsbereichen eine starke Entlastung des Quellcodes und eine Reduzierung der Redundanz. Die Verwendung entsprechender Methoden (z.B.: `getBestMatchingNameDisplay()`) vereinheitlicht das Auftreten und die Übersichtlichkeit der Quellcodeabschnitte. Eine ebenso nützliche Funktion bietet die Validierung von Modellelementen, die an verschiedenen Stellen vorgenommen wird. Tritt ein Problem mit einem Element auf, so liefern entsprechende Methoden einen Fehlercode zurück, der auf der bitweisen Oder-Verknüpfung von Konstanten basiert und eine flexible Möglichkeit bietet, diese Probleme in der Oberfläche hervorzuheben.

### 4.2.4 Annotation

Die Annotation erfolgt durch die Kombination von Taxonomie und Ontologie unter der Verwendung der kompletten Modell-Elemente, die mitunter bisher in den einzelnen Editoren nicht in Betracht gezogen wurden.

**Annotations-Projekt**
Eine Annotation kann nur innerhalb und in Kombination mit einem speziellen Projekt erstellt werden. Dafür ist ein Wizard *AnnotationWizard* von *AbstractWizard* abgeleitet worden, der einen Projektnamen, eine Taxonomie und eine Ontologie bzw. Topic Map

anfordert und daraus ein Projekt mit einem Annotations-Modell erstellt. Ein Projekt kann mit Hilfe von „Natures“ (Beschaffenheiten) ausgezeichnet werden, die eine gesonderte Visualisierung und Behandlung innerhalb des Navigators ermöglichen. Die erstellte *AnnotationNature* wird über den Extension Point „org.eclipse.core.resources.natures“ definiert, über „org.eclipse.ui.ide.projectNatureImages“ mit einem eigenen Icon gekennzeichnet und dem Projekt bei der Erzeugung im Wizard zugewiesen. Weitere Eigenschaften können innerhalb der Implementierung der *Nature* definiert werden, was für das Projekt aber nicht geschehen ist. Nachdem das Annotations-Modell erzeugt wurde, wird der assoziierte Annotations-Editor als Unterklasse von *AbstractEditor* geöffnet. Die Umsetzung des Editors erfolgt nach dem bereits bekanntem Schema und als Einstiegspunkt werden Master- und Detail-Seiten mit Funktionalität bestückt.

**Master- und Detail-Seiten**

Die Master-Seite wird benutzt um die Taxonomie, wie im Taxonomie-Editor, in Baumform darzustellen. Dazu wird der *ContentProvider* aus dem Utility-Plugin der Taxonomie verwendet und leicht modifiziert um das Taxonomie-Modell aus dem Annotations-Modell extrahieren zu können. Der *LabelProvider* der Taxonomie wird ebenfalls verwendet und um Funktionalitäten erweitert, die es ermöglichen den *DocumentDescriptor*-Elementen innerhalb des Viewers Icons zuzuweisen, die den Dateityp des Dateisystems repräsentieren. Dadurch kann der Anwender die erstellten Elemente sofort anhand ihrer bekannten Icons identifizieren. Damit in Verbindung steht die Erzeugung von *DocumentDescriptor*-Elementen über das Kontextmenü (zugänglich gemacht über Extension Points), sowie weitere Funktionen zum Editieren, Entfernen und Öffnen des Objekts mit Hilfe von externen, systemeigenen Anwendungen. Die Dokument-Objekte werden als Blätter an die angegebenen Kategorien angefügt und bilden so einen zweiten Objekttyp innerhalb des Viewers. Daraus resultierend werden zwei verschiedene Detail-Seiten erstellt und je einem Objekt zugewiesen.

Die Detail-Seite für die Kategorie-Objekte wird unverändert, wie aus dem Taxonomie-Plugin, verwendet. Für die Dokument-Objekte wird eine umfassendere, komplexere Detail-Seite erzeugt, in der Annotationen für dieses Objekt bzw. das repräsentierte Dokument erstellt werden können. Die Detail-Seite *AnnotationDetailsPageDocument* besteht aus drei getrennten Bereichen, den Dokument-Eigenschaften, den Dokument-Inhalten und der Annotation. Anstatt diese durch ein Shelf-Widget zusammen zu führen, wurde ein „TabFolder-Widget“ verwendet, welcher die logische und funktionelle Trennung der drei Bereiche stärker hervorheben soll. Die Eigenschaften des Dokuments, extrahiert aus dem Dateisystem, können durch zusätzliche Informationen, wie einem Autor und beliebigen anderen Informationen, angereichert werden. Der Eigenschaften-Bereich bietet die Möglichkeit an, das Dokument und im speziellen die Zusatzinformationen zu editieren, ein neues Dokument neben dem aktuellen einzufügen oder das aktuelle Dokument aus der Taxonomie zu entfernen. Der Inhalts-Bereich ist rein informativ und präsentiert die vom Dokument extrahierten Textstellen. Die Extraktion erfolgt mit Hilfe des Frameworks „Tika“, als erstes externes Framework, was in das System

eingebunden wird. Das Tika Toolkit von Apache erkennt und extrahiert Metadaten und strukturierte Textinhalte aus verschiedenen Dokumententypen durch die Verwendung bestehender Parser-Bibliotheken. Der Annotations-Bereich stellt alle Funktionalitäten bereit, um die Dokumente zu annotieren und inhaltlich in das System zu integrieren. Er ist an den Oberflächenentwurf des MDB angelehnt, ohne diesen selbst zu verwenden, was der Benutzung von *SelectionEvent*s und des *SelectionProvider* des Editors geschuldet ist, welche bei einer Schachtelung von Master-Detail-Blöcken und verschiedenen *SelectionEvent*s nur sehr schwer zu handhaben sind.

Der Aufwand der Implementierung bleibt durch die Verwendung einer *SashForm* mit zwei Container-Komponenten (Composite) überschaubar. Der linke Container fasst die Topics vom Typ „Topic“ und stellt diese baumartig in einem Viewer dar. Er greift dafür auf einen LabelProvider, sowie einen ContentProvider aus dem Utility-Plugin der Topic Map zu und erweitert den ContentProvider um Topic-Elemente von Typ „Instance“, die als Blätter in den Baum eingehangen werden können. Der rechte Container innerhalb des Schiebefensters wird zur Anzeige von Details bezüglich der ausgewählten Topics aus dem linken Container verwendet. Die Ansicht ist äquivalent zur Detail-Seite des Topic Map-Editors und unterscheidet sich nur in den zugewiesenen UI Commands in der Oberfläche. Es können ausschließlich Topics vom Typ „Instance“ erstellt, modifiziert, assoziiert, verschoben und entfernt werden (reguliert über einen *PropertyTester*). Der Zugriff auf die Aktionen erfolgt wie bisher über das Kontextmenü, sowie über Toolbar-Aktionen, die ihre Aufgaben an die jeweiligen *CommandHandler* delegieren. Die Verwendung von imperativen Aktionen anstatt von deklarativen Commands, wie bisher, ist an dieser Stelle durch das Fehlen von aktuellen *SelectionEvents* notwendig um die Aktionen je nach ausgewähltem Objekt zu aktivieren bzw. zu deaktivieren.

**Wizards, Dialoge und AnnotationUtil**
Für die Erstellung eines *DocumentDescriptor*-Elements wird ein Wizard verwendet, der als Eingabe auf der ersten Seite eine URL aus dem Dateisystem benötigt und die Basisinformationen dieser Datei in einer Vorschau anzeigt. Auf der zweiten Seite können die optionalen Eigenschaften des Dokuments erzeugt werden. Der gleiche Wizard wird bei der Modifikation des *DocumentDescriptor* verwendet, wobei die erste Seite ausgeblendet wird und nur die Modifikation der Eigenschaften gestattet ist.

Bei der Erzeugung eines Topic vom Typ „Instance“ (Instanz-Topic) werden der Wizard und die Dialoge zur Erstellung von Topics aus dem Utility-Plugin der Topic Maps verwendet. Zusätzlich wird beim Erstellen automatisch das aktuell ausgewählte Dokument-Objekt als Occurrence für das Instanz-Topic gesetzt, damit die Herkunft nicht manuell hinzugefügt werden muss.

Für das Annotations-Modell wird kein Modell-Werkzeug benötigt, es kann auf die jeweiligen Klassen von Taxonomie und Topic Map zurückgegriffen werden.

### 4.2.5 Benutzer

Die Anmeldung an das System mit Hilfe eines Benutzer-Logins und Passworts wird über den interaktiven *SplashHandler* abgewickelt, der beim Start der Anwendung erscheint. Die Anmeldedaten werden ausgewertet und für die Rolle erlaubte Plugins werden geladen. Existieren zu diesem Zeitpunkt noch keine Benutzer, was beim ersten Start des Systems der Fall ist, so wird ein temporärer Nutzer erzeugt, der initiale Benutzer am System registrieren muss. Es wird überwacht, dass zu jeder Zeit mindestens ein Benutzer mit der Administrator-Rolle vorhanden ist, der die Benutzerverwaltung durchführen kann.

**Master- und Detail-Seiten**
Die Benutzerverwaltung wird mit Hilfe eines Editors vorgenommen, der nach dem Schema des abstrakten Editors *AbstractEditor* mit Master-Details-Block erstellt ist. Auf der Master-Seite wird ein *ControlledViewer* mit einer Baumstruktur initialisiert. Der damit verbundene *UserContentProvider* traversiert das Benutzermodell, indem die vorhandenen Rollen in der ersten Ebene des Baumes angeordnet und die Benutzer, die die jeweilige Rolle innehaben, als Blätter in der zweiten Ebene hinzugefügt werden. Nutzer mit mehreren Rollen werden mehrfach aufgeführt. Der *UserLabelProvider* weist jeder Rolle ein besonderes Icon zur schnellen Identifizierung zu. Benutzer, die mehrere Rollen besitzen, werden über das Icon der Standardrolle innerhalb des Baums dargestellt.

Es werden zwei Detail-Seiten für den Editor registriert, eine für *Role*-Objekte und eine für *User*-Objekte. Die Detail-Seite für die Rollen *UserDetailsPageRole* wird verwendet um Informationen, Restriktionen und Privilegien der entsprechenden Rolle textuell in einer Übersicht anzuzeigen. Die *UserDetailsPage* für die Benutzer besteht aus zwei Sektionen (*Section*), einem rein informativen Benutzerdaten-Abschnitt und einem editierbaren Rollen-Bereich. Beide Bereiche sind vorbereitete Container-Klassen, die auch bei der Generierung eines Benutzers mit Hilfe des Wizard *UserWizard* verwendet werden, der diese Container allerdings auf zwei getrennten Seiten unterbringt um die übersichtliche, schrittweise Eingabe der Benutzerinformationen zu ermöglichen.

**Benutzerdaten und Rollen**
Die zwei Container-Klassen werden von der abstrakten Klasse *AbstractUserComposite* abgeleitet, die gemeinsame Funktionen, wie die Validierung der Eingaben und die Oberflächenadaption zwischen Forms API und normaler SWT API, übernimmt.

Die Benutzerdaten werden über den Container *UserDataComposite* visualisiert. Dazu werden eine Reihe von Text-Widgets verwendet, die mit einem eigenen Marker für Hinweise und Fehleingaben (*ControlDecoration*) ausgezeichnet sind. Über diesen können Informationen zu den einzelnen Feldern angezeigt werden, die Hilfestellungen bei der Eingabe leisten können oder eventuelle Probleme (z.B.: zwei identische Login-Namen) hervorheben. Wird die Komponente mit Hilfe der Forms API initialisiert und damit in-

nerhalb des Editors verwendet, so werden die Text-Widgets für das Editieren gesperrt und die Marker versteckt, damit keine versehentlichen Änderungen an den Nutzerdaten vorgenommen werden.

Die Verwaltung der Nutzerrollen und der Standardrolle erfolgt innerhalb des Containers *UserRoleComposite*. Es werden zwei nebeneinander liegende Listen-Viewer verwendet, die die Zuordnung von Rollen über zwei Button-Widgets verwalten (Zuweisen und Entfernen). Die erste Liste beinhaltet alle Rollen, die dem Benutzer zugeordnet werden können, wobei Rollen, die der Nutzer bereits besitzt durch den verwendeten *UserContentProviderRoles* nicht mehr angezeigt werden. Die zweite Liste bildet das Gegenstück und stellt alle Rollen dar, die der Benutzer derzeit besitzt. Wird ein neuer Nutzer über den Wizard angelegt, so erhält er standardmäßig, durch das Benutzer-Modell festgelegt, die Rolle „Basis“. Jeder Benutzer muss mindestens einer Rolle zugeordnet sein, die nicht entfernt werden kann. Zusätzlich zur Verwaltung der Benutzerrollen, kann der Container verwendet werden, um die Standardrolle des Benutzers festzulegen. Besitzt er nur eine Rolle, so wird diese automatisch als Standard gesetzt, besitzt er mehrere Rollen kann aus diesen eine entsprechende gewählt werden. Wird dem Benutzer die aktuelle Standardrolle entzogen, so erhält er die in der Rangfolge an nächster Stelle[25] stehende Rolle als neue Standardrolle zugewiesen. Auf diese Weise ist gesichert, dass bei versehentlichem Entfernen einer Rolle immer eine aktive Standardrolle beim Nutzer verbleibt.

Das Modell-Werkzeug *UserUtil* kapselt mittels statischer Fabrikmethoden und Validierungen einen kleinen Teil der Modell-Logik und stellt sie externen Klassen bereit.

### 4.2.6 Verwendung von Extension Points

Der besondere Anspruch bei der Entwicklung von Plugins unter Eclipse liegt in der Verwendung von Erweiterungen (Extensions) für bereitgestellte Erweiterungspunkte (Extension Points) – die Einspeisung von Funktionalität in ein bestehendes System über deklarative Angaben. Exemplarische Beispiele sollen die Anwendung dieser Technik für die modellspezifischen Editoren demonstrieren.

#### 4.2.6.1 Editoren

Für die Verwendung eines Editors muss eine Erweiterung für den Erweiterungspunkt „org.eclipse.ui.editors“ erstellt werden, wie in Abbildung 4.6 zu sehen ist.

Die Detailansicht für die Konfiguration der erstellten Extension ist in Abbildung 4.7 abgebildet.

[25] Rangfolge der Benutzerrollen: Administrator, Erweitert, Basis

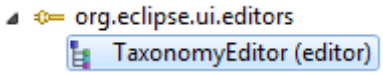


Abbildung 4.6: Exemplarischer Extension Point für Editoren

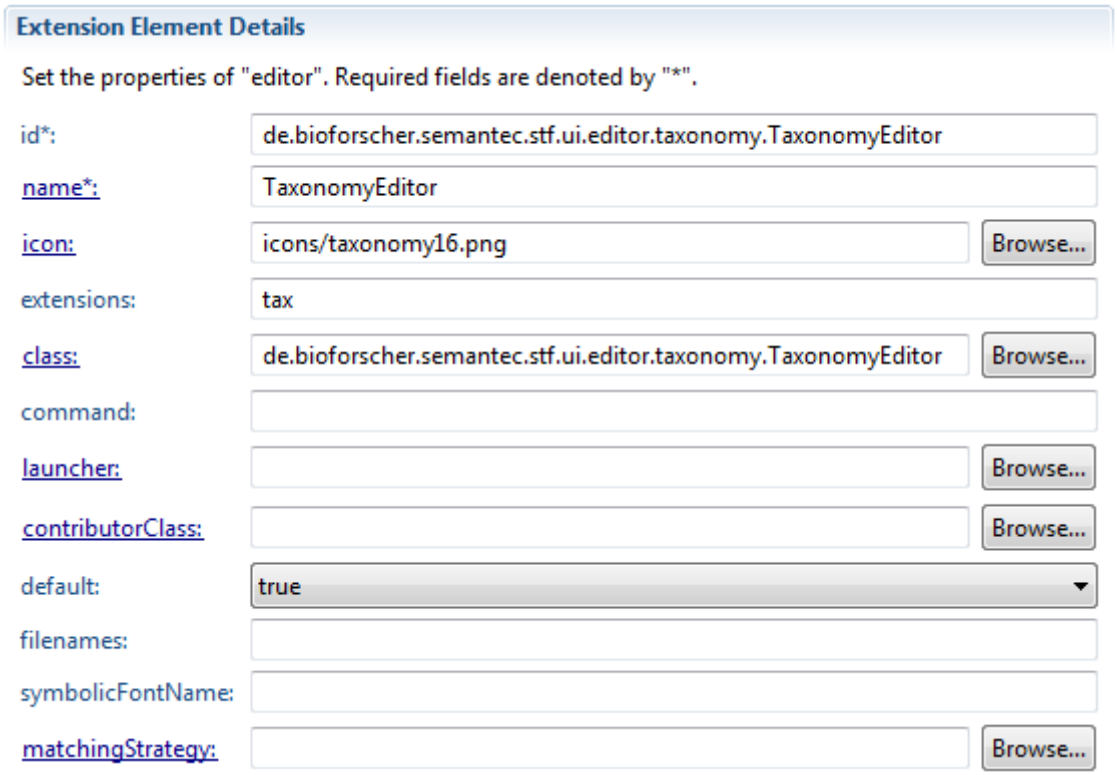


Abbildung 4.7: Exemplarischer Extension Point für Editoren - Detailsansicht

Der Editor erhält eine eindeutige ID (id*), über die er identifiziert wird, im Beispiel des Taxonomie Editors entspricht die ID dem Pfad der implementierten Klasse (class) „de.bioforscher.semantec.stf.ui.editor.taxonomy.TaxonomyEditor“. Die Angabe des Namens (name*) ist, wie die ID, nicht optional und legt den Namen des Editors in der Oberfläche fest. Die Verwendung eines Icons (icon) erleichtert die Unterscheidung verschiedener Editoren. Eine oder mehrere Dateiendungen (extensions), auf die der Editor reagieren soll (in diesem Fall Dateien der Form *.tax), können, durch Kommata separiert, angegeben werden. Alternativ können explizit Dateinamen (filenames) angegeben werden, auf die der Editor reagieren soll. Die Logik des Editors kann mit Hilfe der Angabe einer Klasse (class), eines speziellen Kommandos zum Starten eines externen Editors (command) oder einer Klasse zum Starten eines externen Editors (launcher) erfolgen. Sollen editorbezogene Aktionen in Menüs oder Toolbars eingefügt werden, so kann dies mit Hilfe eines *EditorActionBarContributor* (contributorClass) geschehen. Das Standardverhalten (default) des Editors wird verwendet um beim Öffnen einer Ressource einen entsprechenden Editor zu wählen, um diese Ressource anzuzeigen, falls mehrere Editoren auf eine Dateiendung (oder einen Dateiname) registriert sind. Als Standard gekennzeichnete Editoren werden bei dieser Auswahl bevorzugt. Eine eigene Implementierung des

Auswahl-Algorithmus (matchingStrategy) eines entsprechenden Editors kann ebenfalls angegeben werden. Zusätzlich kann eine definierte Schriftart (symbolicFontName) für den Editor eingestellt werden, wenn die normale Schriftart unzureichend ist.

#### 4.2.6.2 Wizards

Bei der Verwendung eines Wizards können verschiedene Extension Points verwendet werden. Je nach Aufgabe des Wizards, kann dieser zum Exportieren („org.eclipse.ui.exportWizards“), zum Importieren („org.eclipse.ui.importWizards“) oder zum Erzeugen neuer Daten („org.eclipse.ui.newWizards“) klassifiziert werden. Primär wird dadurch die Positionierung innerhalb der Menüeinträge bestimmt.

Die Wizards zur Erzeugung neuer Modelldaten werden dementsprechend als Erweiterung für „org.eclipse.ui.newWizards“ definiert (siehe Abbildung 4.8).

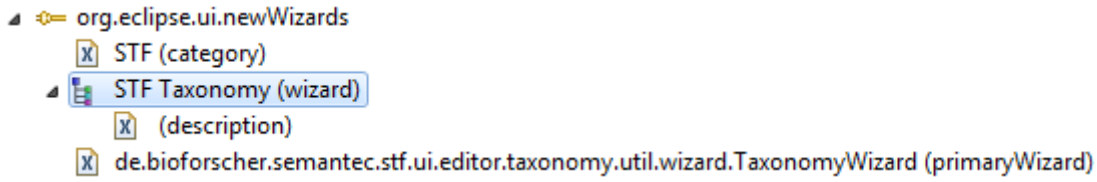


Abbildung 4.8: Exemplarischer Extension Point für Wizards

Neben der Deklaration des Wizards an sich, wird zusätzlich eine Kategorie mit dem Namen „STF“ (category) erzeugt und der Wizard selbst wird über seine ID als primärer Wizard (primaryWizard) referenziert. Primäre Wizards tauchen in Auswahl-Übersichten an hervorstehenden Punkten auf, um direkten Zugriff auf sie zu erhalten. Die Einordnung in eine Kategorie dient der Zusammenfassung projektbezogener Elemente, die für dieses Projekt unter der Kategorie „STF“ aufgeführt werden. Die Deklaration des Wizards an sich (wizard) besitzt ein Unterelement (description) zur Erläuterung der Verwendung des Wizards, welche in der Oberfläche präsentiert wird.

Es ist zusätzlich möglich die Art des Wizards (Export, Import, Neu) für verschiedene Perspektiven über den Extension Point „org.eclipse.ui.perspectiveExtension“ separat festzulegen, sowie über den Extension Point „org.eclipse.ui.navigator.navigatorContent“ einen Schnellzugriff im Kontextmenü des Navigators zu erstellen.

Die Detailansicht für die Konfiguration des Wizards (wizard) ist in der Abbildung 4.9 dargestellt.

Die Angaben von ID (id*), Name (name*), Implementierung (class*) und Icon (icon) sind äquivalent der Angaben bei Editoren. Eine Einordnung in eine Kategorie (category) kann über eine definierte Kategorie-ID erfolgen (wie in Abbildung 4.8 geschehen). Erzeugt der Wizard ein neues Projekt (project), so kann der Wizard zusätzlich in die

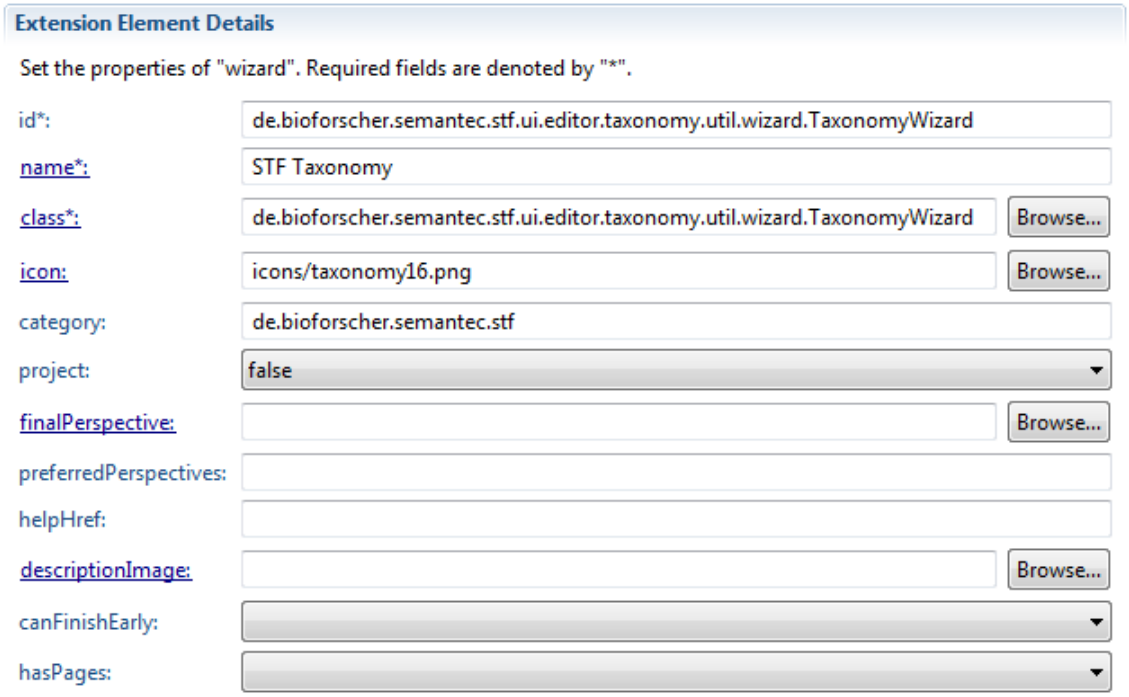


Abbildung 4.9: Exemplarischer Extension Point für Wizards - Detailsansicht

vordefinierte „Neue Projekte“-Kategorie eingeordnet werden. Damit in Verbindung kann eine Perspektive (finalPerspective) angegeben werden, die geöffnet werden soll, sobald dieses Projekt erzeugt wurde. Alternative Perspektiven (preferredPerspectives), die als Arbeits-Perspektiven für das Projekt genehmigt werden, können optional angegeben werden. Erzeugt der Wizard kein Projekt, so sind diese Angaben überflüssig. Eine komplexere Abbildung (descriptionImage) oder eine detaillierte Hilfeseite (helpHref), die mittels einer URL referenziert wird, können den Wizard näher beschreiben. Weiterhin kann definiert werden, ob der Wizard Seiten besitzt (hasPages) und ob er fähig ist ohne das Anzeigen von Seiten fertig gestellt zu werden (canFinishEarly).

# 5 Zusammenfassung und Ausblick

## 5.1 Zusammenfassung

Java wurde speziell wegen der Plattformunabhängigkeit als Programmiersprache für die Entwicklung gewählt. Eine hohe Modularität und Flexibilität wird durch die Verwendung von Eclipse RCP als Basis gewährleistet. Die Einarbeitung und Umsetzung in die verschiedenen Konzepte und Paradigmen der Rich Client Platform von Eclipse konnten zufriedenstellend erreicht werden und bilden eine stabile und gleichzeitig flexible Grundlage für das System. Die Kernelemente für die Entwicklung eines hochgradig anpassbaren Frameworks sind damit etabliert und können nun ausgebaut und erweitert werden.

Eine Evaluierung von Ansätzen zur Repräsentation von Ontologien wurde vorgenommen und die Topic Map wurde dabei als ausdrucksstärkste Variante herausgearbeitet. Sie verfügt über die Möglichkeit eine Menge von Informationen aus verschiedenen Sichten zu betrachten und ermöglicht gleichzeitig die Filterung und Strukturierung von unstrukturierten Daten. Durch den Einsatz von Kontext-Elementen können verschiedene Sprachräume (Multilingualität) abgedeckt werden. Eine Zusammenführung mehrerer Topic Maps aus unterschiedlichen Bereichen kann eine schrittweise Annäherung an eine ganzheitliche Abbildung der Domäne ermöglichen.

Mit Hilfe des Eclipse Modeling Frameworks (EMF) konnten Modelle (Meta-Modelle) erstellt werden, die den Aufbau von Taxonomie und Ontologe detailliert beschreiben. Mit Hilfe des abstrakten Syntax der Meta-Modelle wurden konkrete Modelle und entsprechender Quellcode generiert, der alle benötigten Klassen zur Modellierung und für unterstützende Prozesse enthält. Nach der Erzeugung dieser domänenspezifischen Sprache (DSL) konnte die Oberfläche für den Taxonomie-, Ontologie- und Annotations-Editor entworfen werden. Ein Benutzermodell zur Zugriffssicherung auf das System und Strukturierung der Anwendung nach Funktionen konnte ebenfalls mit Hilfe von EMF erstellt und per Editor editierbar gemacht werden.

Der Prototyp kann einfache, manuelle Annotations-Aufgaben auf den vorhandenen Modellen absolvieren, ist auf verschiedene Systeme portierbar und lässt sich strukturell und funktionell erweitern und anpassen. Für die Anwendungsdomäne der IT-Forensik wurden einige Strukturen bereits vorbereitet, allerdings im aktuellen Stadium der Entwicklung noch nicht integriert.

## 5.2 Ausblick

Die Erweiterung des Systems kann vor allem in den Bereichen der computerlinguistischen Technologien, sowie in einigen architektonischen Grundkonzepten durchgeführt werden.

### 5.2.1 Computerlinguistische Technologien

Ein umfangreicher Textkorpus zur Evaluierung der implementierten Funktionen und zur Entwicklung leistungsfähiger Algorithmen zum (halb-)automatischen Auffinden semantischer Informationen, vor allem in unvollständigen oder fehlerbehafteten Texten, muss erstellt oder bezogen werden. Besonderer Anspruch wird dabei auf die Integration von Slang, Mehrsprachigkeit und verborgenen Bedeutungen gelegt.

Zusätzlich existieren verschiedene Frameworks zur Analyse und Verarbeitung von Dokumenten, die auf eben solche computerlinguistischen Technologien aufbauen. Diese müssen evaluiert und gegebenenfalls in das System integriert werden.

Die Abfrage von annotierten Dokumenten über Filter- und Eingabemasken und die Erstellung von Wissenslandkarten aus den erhaltenen Ergebnissen stellt eine weitere Herausforderung in der Weiterentwicklung dar.

### 5.2.2 Architektur-Anpassungen

Die offene Handhabung der Persistenz durch die Verwendung von EMF, ermöglicht die Datenhaltung auf Datenbanken umzuleiten. Für eine logische Trennung von Arbeitsbereichen können auch mehrere Datenbanken, für zum Beispiel: Metadaten, annotierte Daten und Originaldaten, verwendet werden.

Die OSGi-Implementierung Equinox erlaubt es eine noch flexiblere, serviceorientierte Architektur (SOA) aufzubauen. Eine Weiterentwicklung der Systemarchitektur in diesem Sinne, bringt die Vorteile der modernen Konzepte der SOA (z.B.: lose Kopplung, Wiederverwendbarkeit und modularer Aufbau) in das Framework ein. Die Erzeugung atomarer Service, die Grundfunktionalitäten des Systems kapseln können, können zu komplexeren Servicelandschaften ausgebaut werden und so einen großen, erweiterbaren Funktionsbausatz mit hoher Wiederverwertbarkeit bereitstellen.

# Anhang A: Abbildungen

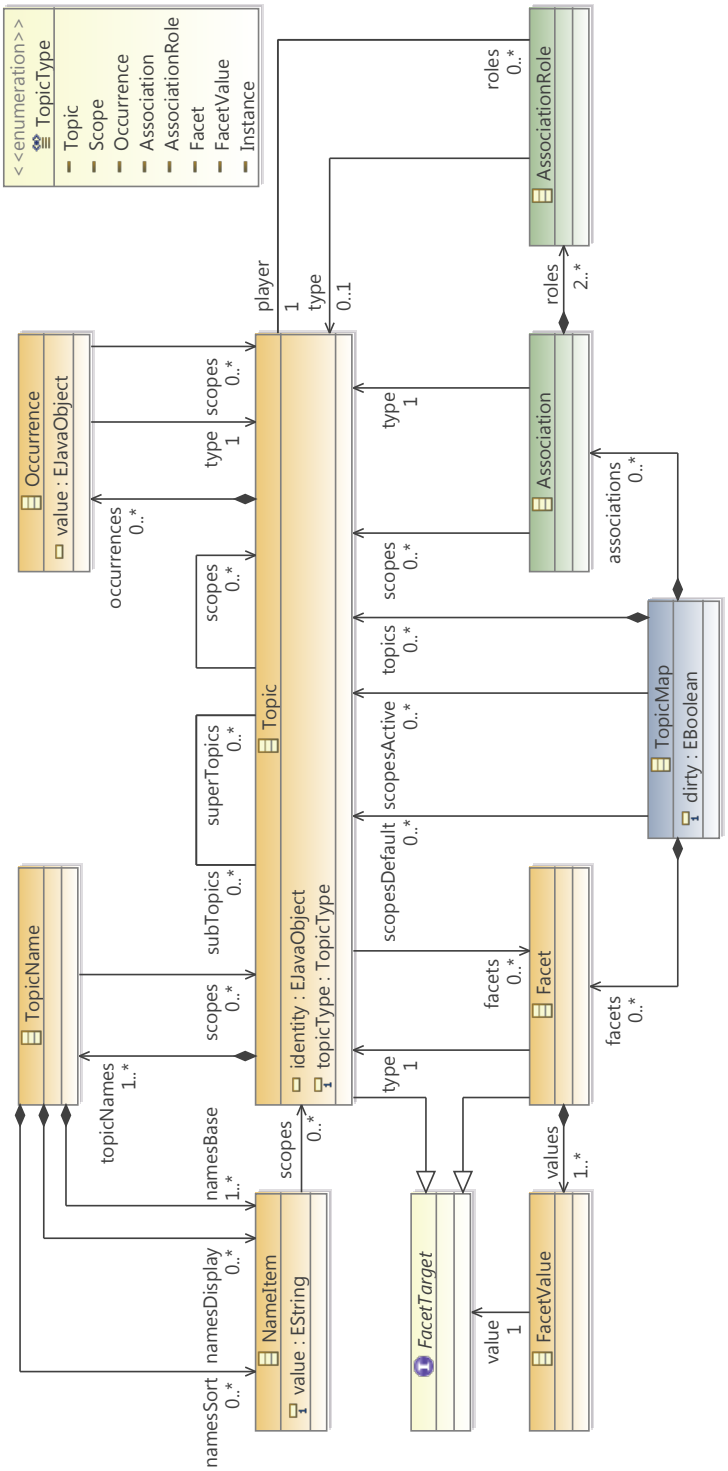


Abbildung A.1: Detaillierte Übersicht über das Topic Map Modell

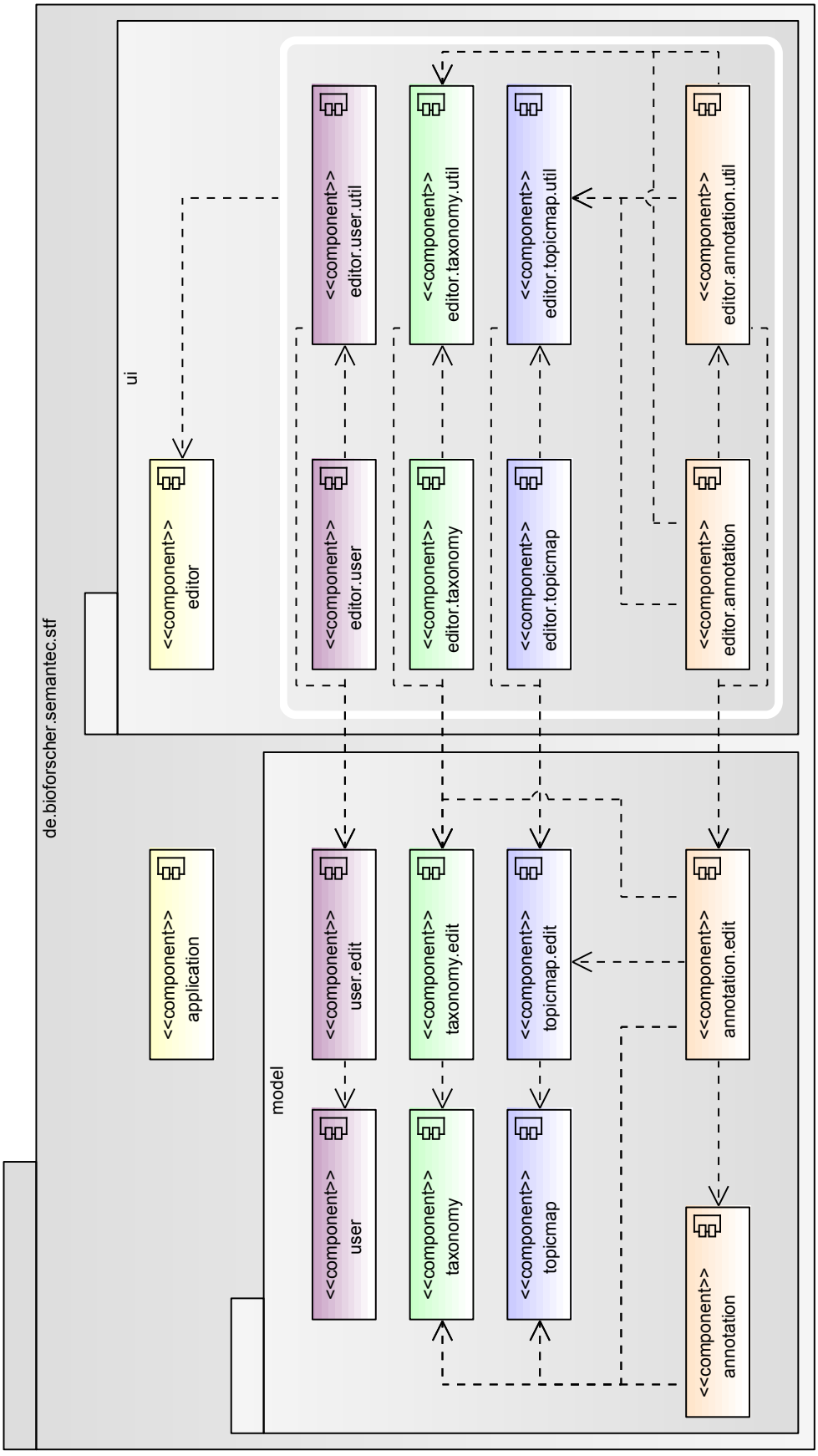


Abbildung A.2: Detaillierte Übersicht über die Abhängigkeiten der erstellten Plugins

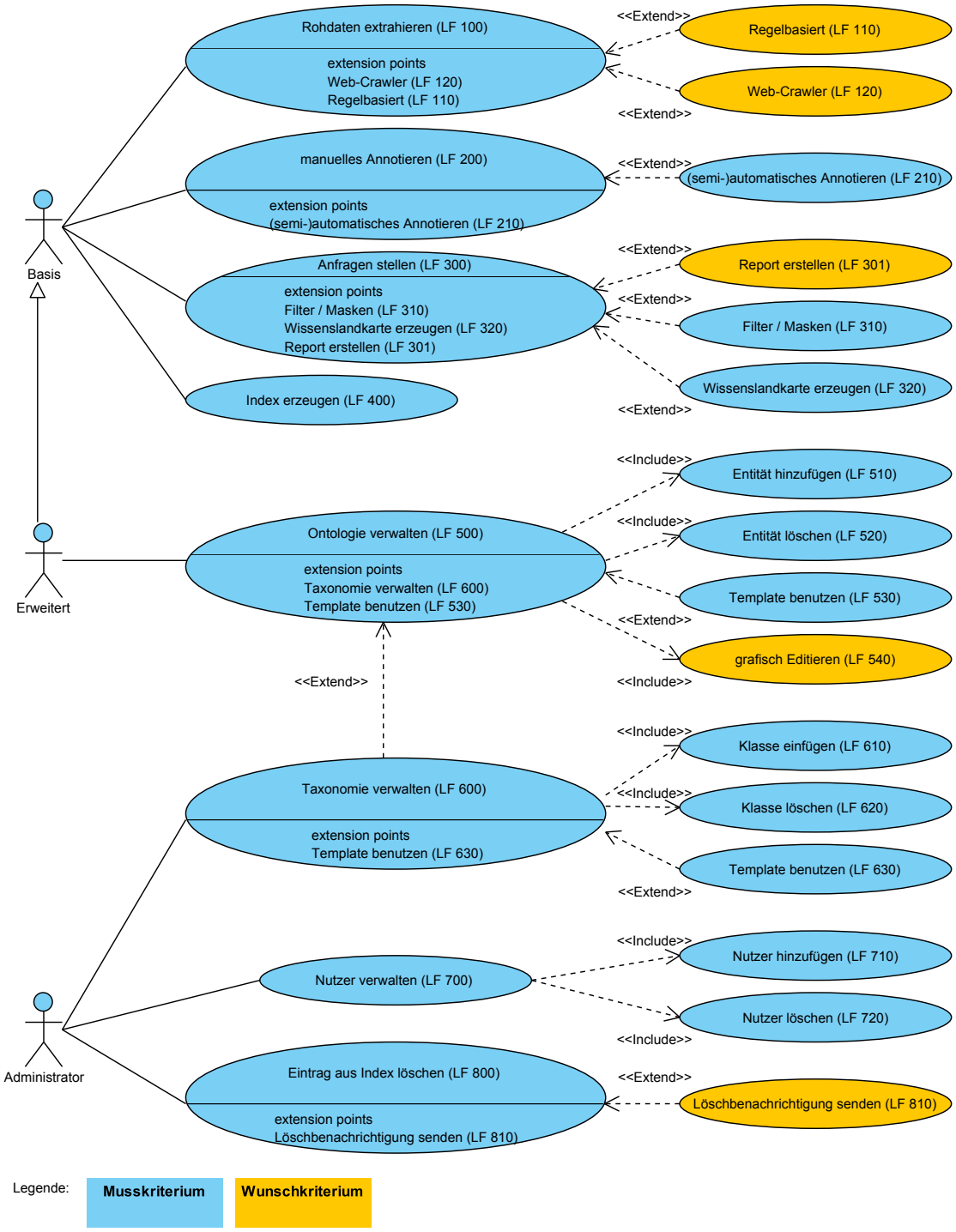


Abbildung A.3: Übersicht der Anwendungsfälle (Use Cases)

# Anhang B: Eclipse-Hausregeln

Die Eclipse-Hausregeln sind Verhaltensregeln, die bei der Entwicklung von Eclipse-Anwendungen beachtet werden sollten. [11]

## B.1 Erweiterer

**Erweiterungsregel (Contribution Rule)** Alles ist eine Erweiterung.

**Konformitätsregel** Erweiterungen müssen erwarteten Schnittstellen entsprechen.

**Regel der gemeinsamen Nutzung (Sharing Rule)** Hinzufügen, nicht ersetzen.

**Nachahmungsregel (Monkey See/Monkey Do Rule)** Beginnen Sie immer damit, die Struktur eines ähnlichen Plug-Ins zu kopieren.

**Relevanzregel** Erweitern Sie nur, wenn es sich vernünftig damit arbeiten lässt.

**Integrationsregel** Integrieren, nicht trennen.

**Verantwortlichkeitsregel** Kennzeichnen Sie deutlich Ihr Plug-In als die Quelle von Problemen.

**Programmiere gemäß API-Vertrag** Prüfen und programmieren Sie entsprechend dem API-Vertrag von Eclipse.

**Other-Regel** Machen Sie alle Erweiterungen verfügbar, wobei diejenigen, die für die aktuelle Perspektive normalerweise nicht relevant sind, in einem OTHER-Dialogfeld erscheinen.

**Anpassen an *IResource*-Regel** Definieren Sie nach Möglichkeit einen *IResource*-Adapter für Ihre Domänenobjekte.

**Schichtenregel** Trennen Sie sprachneutrale von sprachspezifischer Funktionalität und Kernfunktionalität von der Funktionalität der Benutzeroberfläche.

**Benutzerkontinuität (User Continuity Rule)** Bewahren Sie den Zustand der Benutzeroberfläche zwischen zwei Sitzungen.

## B.2 Enabler

**Einladungsregel** Erlauben Sie anderen nach Möglichkeit, Beiträge zu Ihren Erweiterungen zu liefern

**Lazy Loading-Regel (Faules Laden)** Erweiterungen werden nur geladen, wenn sie benötigt werden.

**Sichere Plattform-Regel** Als Anbieter eines Erweiterungspunktes müssen Sie sich selbst gegen Fehlverhalten seitens der Erweiterer schützen.

**Fair Play-Regel** Alle Clients halten sich an die gleichen Regeln. Das gilt auch für mich.

**Regel der expliziten Erweiterung** Deklarieren Sie explizit, wo eine Plattform erweitert werden kann.

**Vielfaltregel (Diversity Rule)** Erweiterungspunkte akzeptieren mehrere Erweiterungen.

**Abschirmungsregel (Good Fences Rule)** Schützen Sie sich selbst, wenn Sie die Steuerung Ihres Codes nach außen abgeben.

**Entscheidungsregel (User Arbitration Rule)** Wenn es mehrere anwendbare Erweiterungen gibt, lässt man den Benutzer entscheiden, welche er verwenden möchte.

**Regel der expliziten API** Trennen Sie die API von den internen Schnittstellen.

**Stabilitätsregel** Nachdem Sie andere zu Erweiterungen eingeladen haben, dürfen Sie die Regeln nicht mehr ändern.

**Regel der defensiven API** Legen Sie nur die API offen, in die Sie Vertrauen haben, seien Sie aber darauf vorbereitet, mehr API offen zu legen, wenn Clients danach verlangen.

## B.3 Veröffentlicher

**Lizenzregel** Stellen Sie mit jeder Erweiterung immer eine Lizenz bereit.

# Literaturverzeichnis

[1] CARSTENSEN, K.-U.: *Sprachtechnologie - Ein Überblick*. Web publication, http://www.kai-uwe-carstensen.de/Publikationen/Sprachtechnologie.pdf, 2012.

[2] CARSTENSEN, K.-U., C. EBERT, C. EBERT, S. JEKAT, R. KLABUNDE und H. LANGER: *Computerlinguistik und Sprachtechnologie - Eine Einführung*. Spektrum Akademischer Verlag, 3. Aufl., 2010.

[3] CLAYBERG, E. und D. RUBEL: *Eclipse Plug-ins*. Addison-Wesley Verlag, 3. Aufl., 2009.

[4] CWALINA, K. und B. ABRAMS: *Richtlinien für das Framework-Design*. Addison-Wesley Verlag, 1. Aufl., 2006.

[5] DENGEL, A.: *Semantische Technologien*. Spektrum Akademischer Verlag, 1. Aufl., 2012.

[6] ECLIPSE: *Eclipse documentation: Activities*. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_advext_activities.htm (Documentation), 2012.

[7] ECLIPSE: *Eclipse documentation: Common Navigator Framework*. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fcnf.htm (Documentation), 2012.

[8] ECLIPSE: *FAQ What is an Eclipse product?*. http://wiki.eclipse.org/FAQ_What_is_an_Eclipse_product%3F (FAQ), 2012.

[9] FOBBE, E.: *Forensische Linguistik - Eine Einführung*. Narr Verlag, 1. Aufl., 2011.

[10] FOWLER, M., K. BECK, J. BRANT, W. OPDYKE und D. ROBERTS: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1. Aufl., 1999.

[11] GAMMA, E. und K. BECK: *Eclipse erweitern: Prinzipien, Patterns und Plug-Ins*. Addison-Wesley Verlag, 1. Aufl., 2004.

[12] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns. Elements of Reusable Object-Oriented Software.*. Addison-Wesley Verlag, 1. Aufl., 1994.

[13] GERSDORF, R.: *Topic Maps zur Strukturierung von eLearning Inhalten*. Proceedings of KnowTech, Dresden, 2003.

[14] GESCHONNEK, A.: *Computer-Forensik: Computerstraftaten erkennen, ermitteln, aufklären*. dpunkt.verlag, 5. Aufl., 2011.

[15] HEARST, M.: *Untangling Text Data Mining*. Proceedings of ACL'99: the 37th Annual Meeting of the Association for Computational Linguistics, 1999.

[16] HEARST, M.: *What Is Text Mining?*. `http://people.ischool.berkeley.edu/~hearst/text-mining.html` (Unpublished essay), 2003.

[17] JTC 1/SC 34/WG 3: *Topic Maps, Information Technology, Document Description and Processing Languages, Technologies de l'information – Cartes topiques*. `http://www.y12.doe.gov/sgml/sc34/document/0322_files/iso13250-2nd-ed-v2.pdf`, 2002.

[18] MEHLER, A. und C. WOLFF: *Einleitung: Perspektiven und Positionen des Text Mining*. Zeitschrift für Computerlinguistik und Sprachtechnologie, Band 20, Heft 1, Regensburg, 2005.

[19] NAVIGLI, R. und P. VELARDI: *Learning Domain Ontologies from Document Warehouses and Dedicated Web Sites*. Computational Linguistics, Vol. 30, No. 2 (MIT Press, 2004): 151–179, 2004.

[20] NEBULA: *Nebula Project*. `http://www.eclipse.org/nebula/`, Juli 2012.

[21] PEPPER, S.: *Navigating haystacks and discovering needles - Introducing the new topic map standard*. Markup Languages: Theory and Practice, Vol. 1, No. 4 (MIT Press, 1999), 1999.

[22] RATH, H. H. und S. PEPPER: *Topic Maps: Introduction and Allegro*. XML 1999, 1999.

[23] REICHERT, S.: *Eclipse RCP im Unternehmenseinsatz*. dpunkt.verlag, 1. Aufl., 2009.

[24] SPRANGER, M., S. SCHILDBACH, F. HEINKE, S. GRUNERT und D. LABUDDE: *Semantic Tools for Forensics: A Highly Adaptable Framework*. IMMM 2012, The Second International Conference on Advances in Information Mining and Management, 2012.

[25] STAHL, T., M. VÖLTER, S. EFFTINGE und A. HAASE: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. dpunkt.verlag, 2. Aufl., 2007.

[26] STEINBERG, D., F. BUDINSKY, M. PATERNOSTRO und E. MERKS: *EMF: Eclipse Modeling Framework*. Addison-Wesley Verlag, 2. Aufl., 2009.

[27] ULLRICH, M., A. MAIER und J. ANGELE: *Taxonomie, Thesaurus, Topic Map, Ontologie - ein Vergleich*. http://www.ullri.ch/download/Ontologien/ttto13.pdf (White Paper), 2004.

[28] USZKOREIT, H.: *Vorlesungen: Einführung in die Computerlinguistik*. Universität des Saarlandes und Deutsches Forschungszentrum für Künstliche Intelligenz http://www.coli.uni-saarland.de/~hansu/vlcl01slides.html (Unpublished lectures), 2002.

[29] VOGEL, L.: *Eclipse Activities – Hide / Display certain UI elements*. http://www.vogella.com/blog/2009/07/13/eclipse-activities/, Juli 2009.

[30] VOGEL, L.: *Eclipse Commands Tutorial*. http://www.vogella.com/articles/EclipseCommands/article.html, Juni 2012.

[31] VOGEL, L.: *Eclipse Extension Points and Extensions - Tutorial*. http://www.vogella.com/articles/EclipseExtensionPoint/article.html, Juni 2012.

[32] VOGEL, L.: *Eclipse Forms API - Tutorial*. http://www.vogella.com/articles/EclipseForms/article.html, Juni 2012.

[33] VOGEL, L.: *Eclipse Modeling Framework (EMF) - Tutorial*. http://www.vogella.com/articles/EclipseEMF/article.html, Juli 2012.

[34] VOGEL, L.: *Eclipse RCP Tutorial*. http://www.vogella.com/articles/Eclipse3RCP/article.html, Mai 2012.

[35] WIDHALM, R. und T. MÜCK: *Topic Maps*. Springer Verlag, 1. Aufl., 2002.

# Glossar

**Annotation** Eine Annotation ist eine beschreibende Zusatzinformation für ein Objekt. Sie hilft bei der Erklärung und beinhaltet ergänzende Informationen.

**Computerlinguistik** Die Computerlinguistik ist das Fachgebiet, das sich mit der maschinellen Verarbeitung natürlicher Sprache beschäftigt. Sie ist im Überschneidungsbereich von Informatik und Linguistik angesiedelt.

**Domain-Framework** Ein Framework bildet einen erweiterbaren Anwendungsrahmen für ein bestimmtes Problem bzw. einen Problembereich. Ein Domain-Framework bietet Funktionen und Strukturen an, die zur Lösung dieses Problembereichs benötigt werden.

**DRY-Prinzip** Das DRY-Prinzip (Don't Repeat Yourself) hat zum Ziel möglichst keine oder nur sehr geringe Redundanz aufkommen zu lassen.

**Entität** Eine Entität entspricht einem Objekt, ohne eigene Funktionen, Eigenschaften und Methoden.

**extrinsisch** Extrinsisch bedeutet von außen her (angeregt), nicht aus eigenem Antrieb erfolgend.

**Factory** Eine Factory (engl. Fabrik) ist eine Klasse, die statische Methoden bereit stellt, welche instanziierte Objekte liefern. Die Aufgabe einer Factory ist es Objekte zu erzeugen.

**IT-Forensik** Die IT-Forensik (Computer-Forensik) oder auch Digitale Forensik wird definiert als der Nachweis und die Aufklärung von strafbaren Handlungen z.B. durch Analyse von digitalen Spuren.

**Klasse** Eine Klasse beschreibt die Struktur und das Verhalten eines Objektes.

**Konzeptualisierung** Eine Konzeptualisierung ist der Versuch, ein Phänomen der realen Welt auf eine abstrakte Art und Weise darzustellen. Dabei darf nur der relevante Kern des Phänomens betrachtet werden.

**Linguistik** Linguistik (Sprachwissenschaft) ist eine Wissenschaft, die in verschiedenen Herangehensweisen die menschliche Sprache untersucht.

**Modul** Ein Modul ist ein funktionsorientierter Teil eines größeren Systems.

**Objekt** Ein Objekt ist ein existierendes Ding aus der Umwelt und besitzt in der Objektorientierten Programmierung eine Reihe ihm eigenen Funktionen, Eigenschaften und Methoden.

**Ontologie** Eine Ontologie stellt ein Netzwerk von Informationen mit logischen Relationen dar.

**Paradigma** Ein Paradigma beschreibt eine bestimmte wissenschaftliche Lehrmeinung, Denkweise oder Art der Weltanschauung, die sich über einen Zeitraum hinweg etabliert hat.

**Persistenz** Persistenz ist die Fähigkeit Daten oder Objekte auf nicht-flüchtigen Medien (z.B.: Dateisystem, Datenbank) zu speichern.

**Redundanz** Redundanz ist das mehrfache Vorkommen des selben Sachverhaltes (z.B.: funktionell oder strukturell identische Objekte), wobei die Wiederholung als nicht notwendig eingeordnet wird.

**Refactoring** Refactoring ist eine Methode zur Restrukturierung eines Programmstücks, bei der die interne Struktur verändert wird, ohne eine Veränderung des Verhaltens nach Außen hin.

**Schnittstelle** Die Schnittstelle bildet einen Teil des Systems, der der Kommunikation dient. Über sie können Teilsysteme miteinander kommunizieren.

**Taxonomie** Eine Taxonomie ist ein einheitliches Modell, um Objekte eines Bereiches nach bestimmten Kriterien zu klassifizieren.

**Toolbox** Eine Toolbox beschreibt eine Auswahl an Werkzeugen (Tools) zur Bearbeitung von Problemstellungen.

**Widget** Ein Widget (zusammengesetzt aus Wi(ndow) und (Ga)dget) ist eine funktionelle Komponente einer grafischen Oberfläche.

**Wizard** Ein Wizard ist ein modales Fenster, ähnlich einem Dialog mit mehreren Seiten.

# Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 19. Dezember 2012